NoiStudiamo

DISPENSA DI TEORIA

Basi di Dati
Teoria e metodi

Modello relazionale, algebra, SQL, progettazione, normalizzazione, organizzazione fisica, transazioni, XML ed esercizi svolti.

A CURA DI

Damiano La Pusata

Ottobre 2026

INDICE

Indice dei capitoli

I capitoli della dispensa, con il numero della pagina di inizio. Tocca un capitolo per andarci.

1. Introduzione alle basi di dati 3
2. Il modello relazionale 6
3. Algebra relazionale 9
4. Ottimizzazione algebrica delle interrogazioni 15
5. SQL: le interrogazioni 18
6. SQL: definizione dei dati, viste e controllo dell'accesso 22
7. Basi di dati attive: i trigger 26
8. Progettazione concettuale e modello ER 29
9. Progettazione logica 33
10. Normalizzazione 37
11. Organizzazione fisica e indici 44
12. Transazioni e affidabilità 49
13. Controllo della concorrenza 53
14. Basi di dati per XML 57
15. Esercizi svolti 60

CAPITOLO 1

Introduzione alle basi di dati

Che cosa sono una base di dati e un DBMS, perché si descrivono i dati a più livelli e quali linguaggi si usano per definirli e interrogarli.

Sistema informativo e sistema informatico

Ogni organizzazione (un'impresa, una banca, un ospedale, un'università) è un **sistema organizzativo**: un insieme di risorse, cioè persone, denaro, beni e informazioni, e di regole che coordinano le attività verso uno scopo. La parte che raccoglie, conserva, elabora e scambia le informazioni è il **sistema informativo**. Il concetto è indipendente dal computer: l'anagrafe di un comune o i registri di una banca esistevano da secoli su carta. La porzione automatizzata del sistema informativo, quella che gestisce le informazioni con la tecnologia informatica, è il **sistema informatico**.

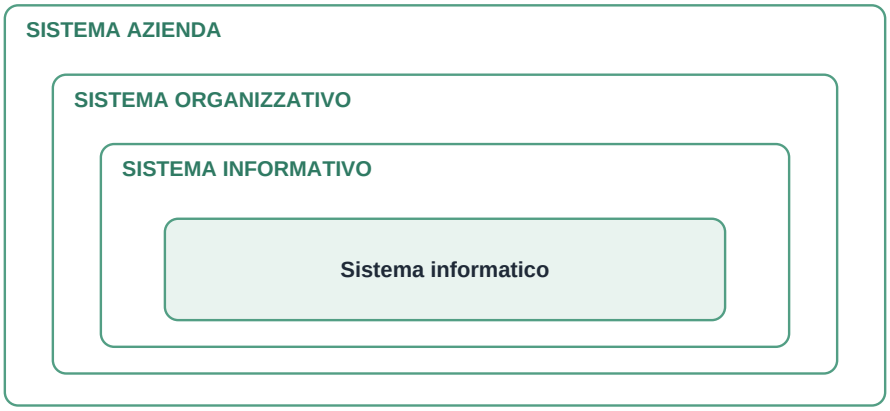


Figura 1.1. Ogni livello è contenuto nel precedente: il sistema informatico è solo la parte automatizzata del sistema informativo.

Nei sistemi informatici le informazioni sono rappresentate in modo essenziale attraverso i **dati**. Un dato da solo non dice nulla: sul foglio con scritto "Mario" e "2075" ci sono due dati, che diventano informazione solo se sappiamo che rispondono alla domanda "a chi mi rivolgo per il problema X e qual è il suo interno telefonico?". I dati sono una risorsa strategica perché cambiano molto più lentamente dei processi, delle tecnologie e dei ruoli delle persone.

DEFINIZIONE

Una **base di dati** è un insieme organizzato di dati, che esistono ed evolvono nel tempo, utilizzati da un'organizzazione per lo svolgimento delle proprie attività.

Il DBMS

Un **DBMS** (DataBase Management System) è il software che gestisce collezioni di dati con tre caratteristiche che i normali programmi non sanno trattare insieme:

- **grandi**: molto più grandi della memoria centrale, quindi stanno in memoria secondaria;
- **persistenti**: la loro vita non dipende dalle singole esecuzioni dei programmi che li usano;
- **condivise**: usate da applicazioni e utenti diversi, anche contemporaneamente.

Per gestirle il DBMS garantisce anche **affidabilità** (i dati resistono ai guasti hardware e software), **privatezza** (ogni utente accede solo a ciò per cui è autorizzato), **efficienza** (uso accorto di tempo e spazio) ed **efficacia** (rende produttivo il lavoro di chi lo usa). Esempi di DBMS relazionali sono MySQL, PostgreSQL, Oracle, SQL Server e

DB2.

La condivisione ha un prezzo: attività diverse su dati comuni richiedono **meccanismi di autorizzazione**, e l'accesso contemporaneo di più utenti richiede il **controllo della concorrenza**. Senza un DBMS, ogni ufficio terrebbe la propria copia dei dati, con **ridondanza** (informazioni ripetute) e quindi **rischio di incoerenza** (copie che non coincidono più).

VANTAGGI	SVANTAGGI
Dati come risorsa comune e modello unico della realtà	Costo dei prodotti e della transizione
Gestione centralizzata, standardizzazione, economia di scala	Architetture complesse anche per esigenze minime
Riduzione di ridondanze e incoerenze	Prestazioni non ottimali per applicazioni molto specifiche
Indipendenza dei dati: sviluppo e manutenzione più semplici	

Modelli dei dati, schemi e istanze

Un **modello dei dati** è un formalismo composto da una notazione per descrivere i dati e da un insieme di operatori per manipolarli. Il suo cuore sono i **costruttori di tipo**, come nei linguaggi di programmazione: il modello relazionale, per esempio, ha il costruttore *relazione*, che definisce insiemi di record omogenei. Si distinguono due famiglie:

- **modelli logici**: usati dai DBMS per organizzare i dati e dai programmi per accedervi; sono astratti ma riflettono un'organizzazione precisa (relazionale, gerarchico, reticolare, a oggetti);
- **modelli concettuali**: descrivono i concetti del mondo reale indipendentemente da qualsiasi sistema, e si usano nelle prime fasi della progettazione; il più noto è il modello **Entità Relazione** (ER).

In ogni base di dati si distingue lo **schema**, la parte invariante nel tempo che descrive la struttura (per esempio l'intestazione di una tabella), dall'**istanza**, i valori attuali, che cambiano continuamente (le righe). Lo schema è l'aspetto *intensionale*, l'istanza quello *estensionale*.

Architettura a tre livelli ANSI/SPARC

Lo standard ANSI/SPARC descrive i dati su tre livelli, ciascuno con il proprio schema:

- **schema logico** (o concettuale nel lessico ANSI): descrive l'intera base di dati nel modello logico del DBMS;
- **schema interno** (o fisico): rappresenta lo schema logico con strutture di memorizzazione concrete, cioè file, indici, blocchi;
- **schemi esterni**: descrivono porzioni della base di dati come le vede un certo utente o applicazione, tipicamente attraverso **viste**.

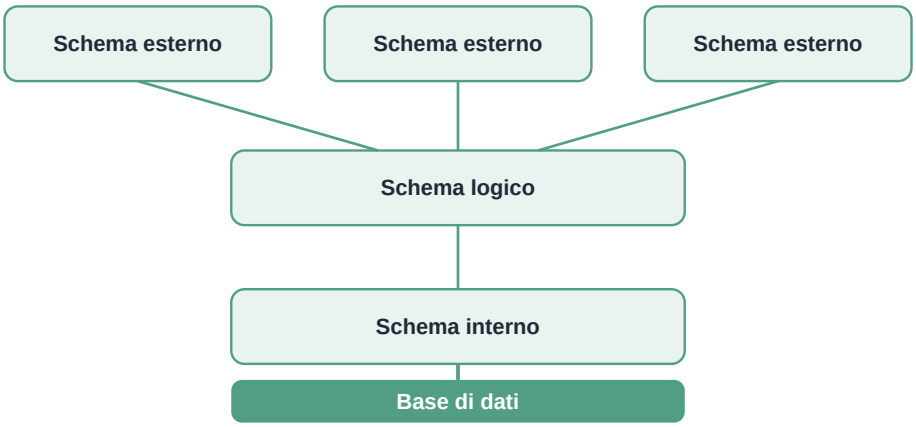


Figura 1.2. Architettura ANSI/SPARC: gli utenti accedono attraverso gli schemi esterni, che si appoggiano allo schema logico, realizzato a sua volta dallo schema interno.

L'articolazione in livelli garantisce l'**indipendenza dei dati**, in due forme:

- **indipendenza fisica**: i livelli logico ed esterno non dipendono da quello fisico; una tabella si usa nello stesso modo qualunque sia la sua realizzazione, che può cambiare senza modificare i programmi;
- **indipendenza logica**: il livello esterno non dipende da quello logico; si possono aggiungere o modificare viste senza toccare lo schema logico, e le modifiche allo schema logico che lasciano invariato lo schema esterno sono trasparenti.

ANALOGIA

Pensa a un'automobile: il guidatore usa volante e pedali (livello esterno) senza sapere se il motore è a benzina o elettrico (livello fisico). Il costruttore può cambiare il motore e il guidatore continua a guidare nello stesso modo: questa è l'indipendenza fisica.

Linguaggi e utenti

Un DBMS offre due categorie di linguaggi, che in SQL convivono nello stesso linguaggio:

- **DDL** (Data Definition Language): definisce gli schemi logici, esterni e fisici, e le autorizzazioni (CREATE TABLE, CREATE VIEW, CREATE INDEX, GRANT);
- **DML** (Data Manipulation Language): interroga e modifica le istanze (SELECT, INSERT, UPDATE, DELETE).

Gli utenti si dividono in amministratore della base di dati (DBA), che ne cura progetto, prestazioni e sicurezza; progettisti e programmatori, che realizzano schemi e applicazioni; utenti finali, che usano le applicazioni oppure interrogano direttamente la base di dati.

CAPITOLO 2

Il modello relazionale

Il modello introdotto da Codd nel 1970: relazioni, schemi, valori nulli e i vincoli di integrità, con particolare attenzione a chiavi e chiavi esterne.

Relazione matematica e relazione del modello

Il termine relazione ha tre significati che conviene tenere distinti: la relazione **matematica** della teoria degli insiemi, la relazione del **modello relazionale** (una tabella) e la **relationship** del modello ER, che in italiano chiameremo **associazione** proprio per evitare confusione.

DEFINIZIONE

Dati n insiemi $D_1, \ldots, D_n$ (i **domini**, anche non distinti), il **prodotto cartesiano** $D_1 \times \ldots \times D_n$ è l'insieme di tutte le ennuple $(d_1, \ldots, d_n)$ con $d_i \in D_i$. Una **relazione matematica** su quei domini è un sottoinsieme del prodotto cartesiano.

Essendo un insieme, una relazione non ha ordinamento fra le ennuple e non contiene ennuple ripetute; ogni ennupla invece è ordinata, perché l'i esimo valore viene dall'i esimo dominio. Questa struttura **posizionale** è scomoda: in Partite(Juve, Lazio, 3, 1) il significato di ogni valore dipende dalla posizione. Il modello relazionale associa allora a ogni dominio un nome, l'**attributo**, che ne descrive il ruolo (Casa, Fuori, RetiCasa, RetiFuori): la struttura diventa **non posizionale** e l'ordine delle colonne non conta più.

Schemi e istanze

Un **tipo ennupla** di grado n è $(A_1: T_1, \ldots, A_n: T_n)$, con A_i attributi e T_i tipi primitivi (interi, reali, booleani, stringhe). Se T è un tipo ennupla, {T} è un **tipo relazione**, cioè un insieme di ennuple di tipo T. Uno **schema di relazione** R: {T} è una coppia nome e tipo relazione; uno **schema relazionale** (o schema di base di dati) è un insieme di schemi di relazione insieme ai vincoli di integrità. Quando i tipi non interessano si usa la forma abbreviata:

Studenti(Nome, Matricola, Indirizzo, Telefono) Esami(Corso, Matricola, Voto)

Un'**istanza** di relazione è un insieme finito di ennuple del tipo indicato; il numero di ennuple è la **cardinalità**, il numero di attributi è il **grado**. L'insieme dei valori ammessi per un attributo è il suo dominio dom(A): per esempio dom(Voto) = {18, …, 30}.

Valori nulli

Non sempre si conosce ogni informazione: il telefono di uno studente può mancare perché non esiste, perché esiste ma è ignoto, o perché non si sa nemmeno se esista. Il modello relazionale usa un unico valore speciale, **NULL**, estraneo a tutti i domini, che segnala l'assenza di informazione. I nulli vanno limitati: una relazione piena di NULL è poco utile, e un nullo in un attributo usato per identificare le ennuple è un problema serio, come vedremo con le chiavi.

Vincoli di integrità

Non tutte le istanze sintatticamente corrette rappresentano situazioni reali: un voto 32, una lode con voto 27, due studenti con la stessa matricola. Un **vincolo di integrità** è un predicato che ogni istanza deve soddisfare; un'istanza è **valida** (o legale) se li soddisfa tutti. I vincoli migliorano la qualità dei dati, arricchiscono il significato dello schema e sono sfruttati dal DBMS anche per ottimizzare le interrogazioni.

- **vincoli intrarelazionali**: riguardano una sola relazione; i più importanti sono NOT NULL, i vincoli di ennupla (come 18 ≤ Voto ≤ 30, oppure Lode solo se Voto = 30) e i vincoli di chiave;
- **vincoli interrelazionali**: coinvolgono più relazioni; il più importante è il vincolo di integrità referenziale, espresso con le chiavi esterne.

Superchiavi, chiavi e chiave primaria

DEFINIZIONE

Un insieme X di attributi di R è una **superchiave** se nessuna istanza valida contiene due ennuple distinte t_1 e t_2 con $t_1[X] = t_2[X]$.

Una **chiave** è una superchiave **minimale**: nessun suo sottoinsieme proprio è a sua volta una superchiave.

In Studenti, {Nome, Matricola} è una superchiave ma non una chiave, perché contiene la superchiave {Matricola}. Attenzione: le chiavi si stabiliscono dalla **semantica** dell'applicazione, non da un'istanza. In una certa istanza nessun nome si ripete, ma due studenti omonimi sono perfettamente possibili, quindi {Nome} non è una chiave. Poiché una relazione non contiene ennuple uguali, l'insieme di tutti gli attributi è sempre una superchiave: ogni relazione ha quindi **almeno una chiave**.

Le chiavi garantiscono l'accessibilità a ogni singolo dato e permettono di collegare dati in relazioni diverse: il modello relazionale è **basato su valori**, non su puntatori. Con i valori nulli però una chiave non identifica più le ennuple e non permette di riferirle. Per questo, fra le chiavi se ne sceglie una, la **chiave primaria**, su cui non sono ammessi valori nulli; si indica sottolineandola.

Chiavi esterne e integrità referenziale

DEFINIZIONE

Un insieme di attributi $\{A_1, \ldots, A_n\}$ di R è una **chiave esterna** che riferisce la chiave primaria $\{B_1, \ldots, B_n\}$ di S se, in ogni istanza valida, per ogni ennupla r di R (con valori non nulli su quegli attributi) esiste un'ennupla s di S tale che $r.A_i = s.B_i$ per ogni i.

In Esami(Corso, Matricola, Voto), Matricola è chiave esterna verso Studenti e Corso è chiave esterna verso Corsi: non si può registrare un esame di uno studente o di un corso inesistente. Nella notazione grafica le chiavi esterne si indicano spesso con una sottolineatura tratteggiata o in corsivo. Quando la chiave primaria riferita è composta, anche la chiave esterna è composta e l'ordine degli attributi conta: in Infrazioni(Codice, Data, Vigile, Prov, Numero) la coppia (Prov, Numero) riferisce Auto(Prov, Numero).

METODO: INDIVIDUARE CHIAVI PRIMARIE ED ESTERNE

Dato uno schema relazionale, chiavi primarie ed esterne si individuano con un procedimento meccanico:

- per ogni relazione chiediti quale attributo, o combinazione, identifica una sola ennupla secondo il significato reale (un codice, un id, oppure la combinazione delle chiavi delle entità collegate);
- ogni attributo che ha lo stesso nome o lo stesso ruolo della chiave primaria di un'altra relazione è una chiave esterna verso di essa: indica sempre la relazione e l'attributo riferiti;
- nelle relazioni che rappresentano associazioni molti a molti la chiave primaria è di solito la combinazione delle chiavi esterne, a volte arricchita da una data se lo stesso legame può ripetersi nel tempo.

Esempio: in Noleggio(cliente, auto, data_inizio, km_percorsi) la chiave primaria è (cliente, auto, data_inizio), perché lo stesso cliente può noleggiare la stessa auto più volte; cliente riferisce Cliente(codice) e auto riferisce Auto(targa).

Esempio svolto: un autonoleggio

Si considerino le relazioni Clienti(Codice, Cognome, Nome, Telefono), Auto(Targa, Modello, Sede), Noleggi(Auto, Inizio, Fine, Cliente), Dipendenti(Matricola, Cognome, Sede) e Sedi(Codice, Citta, Responsabile). Si chiede di individuare chiavi, vincoli di integrità referenziale e attributi che possono essere nulli.

RELAZIONE	CHIAVE PRIMARIA	CHIAVI ESTERNE	POSSIBILI NULLI
Clienti	Codice	nessuna	Telefono, se non comunicato
Auto	Targa	Sede → Sedi(Codice)	nessuno
Noleggi	(Auto, Inizio)	Auto → Auto(Targa); Cliente → Clienti(Codice)	Fine, per i noleggi in corso
Dipendenti	Matricola	Sede → Sedi(Codice)	nessuno
Sedi	Codice	Responsabile → Dipendenti(Matricola)	Responsabile, se il posto è vacante

Due osservazioni utili. La chiave di Noleggi non può essere la sola Auto, perché la stessa auto viene noleggiata molte volte: serve la data di inizio, che distingue i noleggi (un'auto non può avere due noleggi che iniziano nello stesso istante). Fra Dipendenti e Sedi i riferimenti sono **reciproci** (il dipendente lavora in una sede, la sede ha un responsabile che è un dipendente): è lecito, ma all'inserimento iniziale uno dei due riferimenti deve poter restare temporaneamente nullo, oppure il controllo va differito alla fine della transazione.

CAPITOLO 3

Algebra relazionale

Il linguaggio procedurale di riferimento del modello relazionale: operatori primitivi e derivati, trattamento dei nulli, viste e gli schemi risolutivi per scrivere le interrogazioni.

L'**algebra relazionale** è un insieme di operatori che prendono relazioni e restituiscono relazioni; essendo chiusi rispetto alle relazioni, si possono comporre in espressioni complesse. È un linguaggio **procedurale**: l'espressione descrive anche l'ordine delle operazioni, a differenza di SQL che è dichiarativo. Notazione: R, S indicano relazioni, A, B attributi, X, Y insiemi di attributi.

Operatori primitivi

Gli operatori primitivi sono sei: ridenominazione, unione, differenza, proiezione, selezione (o restrizione) e prodotto cartesiano. Tutti gli altri si possono esprimere a partire da questi.

OPERATORE	SIMBOLO	RISULTATO
Ridenominazione	$\delta_{A \to B}(R)$	R con l'attributo A rinominato B; cambia solo lo schema
Unione	$R \cup S$	ennuple che stanno in R oppure in S; R e S dello stesso tipo
Differenza	$R - S$	ennuple di R che non stanno in S; R e S dello stesso tipo
Proiezione	$\pi_X(R)$	tutte le ennuple di R ristrette agli attributi X, senza duplicati
Selezione	$\sigma_F(R)$	le ennuple di R che soddisfano la formula F; stesso schema di R
Prodotto	$R \times S$	tutte le combinazioni di una ennupla di R con una di S; attributi disgiunti

Unione e differenza richiedono relazioni con gli **stessi attributi**: se le colonne hanno nomi diversi ma significato compatibile (Padre e Madre, entrambi genitori) si ridenomina prima di unire:

$$\delta_{\text{Padre} \to \text{Genitore}}(\text{Paternità}) \cup \delta_{\text{Madre} \to \text{Genitore}}(\text{Maternità})$$

La **proiezione** taglia verticalmente la tabella: $\pi_{A1,\ldots,An}(R) = \{t[A_1, \ldots, A_n] \mid t \in R\}$. Poiché il risultato è un insieme, la sua cardinalità può essere minore di quella di R se sparisce la chiave e compaiono duplicati. La **selezione** taglia orizzontalmente: $\sigma_F(R) = \{t \mid t \in R \wedge F(t)\}$, dove F è una formula proposizionale costruita con atomi A θ B o A θ c (θ ∈ {=, ≠, <, >, ≤, ≥}) e i connettivi ∧, ∨, ¬. Il **prodotto cartesiano** $R \times S = \{tu \mid t \in R \wedge u \in S\}$ ha come cardinalità il prodotto delle cardinalità e da solo è raramente utile: diventa utile quando lo si fa seguire da una selezione.

Operatori derivati: intersezione e join

L'**intersezione** si ottiene dalla differenza: $R \cap S = R - (R - S)$. Il **join** (giunzione) è l'operatore più importante, perché combina ennuple di relazioni diverse in base ai valori.

JOIN NATURALE

Siano R con attributi XY e S con attributi YZ (Y sono gli attributi comuni). Il **join naturale** $R \bowtie S$ è la relazione su XYZ formata dalle ennuple t tali che $t[XY] \in R$ e $t[YZ] \in S$: si combinano le ennuple che hanno valori uguali sugli attributi con lo stesso nome.

Il **theta join** $R \bowtie_F S$ è un prodotto cartesiano seguito da una selezione, $\sigma_F(R \times S)$; quando F è una congiunzione di uguaglianze si parla di **equi join**. Il join naturale si ottiene ridenominando gli attributi comuni di S, facendo l'equi join sulle coppie corrispondenti e proiettando su XYZ. Il join è commutativo e associativo, quindi si possono scrivere catene $R \bowtie S \bowtie T$ senza ambiguità. Due casi limite: se R e S non hanno attributi comuni il join naturale coincide con il prodotto cartesiano; se hanno tutti gli attributi in comune coincide con l'intersezione.

Le ennuple che non trovano corrispondenza nell'altro operando restano escluse dal risultato e si dicono **dangling**; il join è **completo** se nessuna ennupla resta esclusa. La cardinalità di $R \bowtie S$ è compresa fra 0 e $|R| \cdot |S|$; se il join avviene sulla chiave di S è al più $|R|$, e se c'è anche il vincolo di integrità referenziale da R a S è esattamente $|R|$.

JOIN ESTERNI E SEMI JOIN

Il **join esterno** conserva le ennuple dangling completandole con NULL. Ne esistono tre varianti: **sinistro** $R \bowtie_{LEFT} S$ (conserva le dangling di R), **destro** $R \bowtie_{RIGHT} S$ (quelle di S) e **completo** $R \bowtie_{FULL} S$ (entrambe). Formalmente, con R su XY e S su YZ:

$$R \bowtie_{LEFT} S = (R \bowtie S) \cup ((R - \pi_{XY}(R \bowtie S)) \times \{Z = NULL\})$$

Il **semi join** $R \ltimes S = \pi_{XY}(R \bowtie S)$ restituisce le ennuple di R che partecipano al join, senza aggiungere colonne di S: per esempio Studenti $\ltimes$ Esami dà gli studenti che hanno sostenuto almeno un esame. L'**unione esterna** estende entrambe le relazioni con le colonne dell'altra, riempite di NULL, e poi le unisce.

Divisione

QUOZIENTE

Siano R su XY e S su Y. La **divisione** $R \div S$ è la relazione su X che contiene le ennuple w tali che, per ogni $s \in S$, la combinazione ws sta in R: $R \div S = \{w \mid \{w\} \times S \subseteq R\}$.

La divisione risponde alle domande con **"tutti"**: gli studenti che hanno superato *tutti* i corsi di un elenco, i clienti che hanno ordinato da *tutti* gli agenti. È un operatore derivato:

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

La lettura è intuitiva: $\pi_X(R) \times S$ sono tutte le combinazioni che dovrebbero esserci; togliendo quelle che ci sono davvero $(- R)$ restano quelle mancanti; chi ha almeno una combinazione mancante viene scartato.

Selezione e valori nulli

Una condizione atomica su un attributo nullo non è né vera né falsa: in $\sigma_{Età>40}(Impiegati)$ un impiegato con età NULL viene escluso. Di conseguenza $\sigma_{Età>30}(P) \cup \sigma_{Età\leq30}(P)$ non restituisce tutta P, e lo stesso vale per $\sigma_{Età>30 \vee Età\leq30}(P)$, perché ogni condizione atomica è valutata separatamente. Per riferirsi ai nulli esistono le condizioni apposite **IS NULL** e **IS NOT NULL**:

$$\sigma_{(Età>30) \vee (Età\leq30) \vee (Età\ IS\ NULL)}(Persone) = Persone$$

Viste

Una **vista** è una relazione derivata, il cui contenuto è funzione di altre relazioni ed è definito da un'espressione; le relazioni con contenuto autonomo sono dette di base. Le viste realizzano gli schemi esterni e possono essere:

- **materializzate**: memorizzate, quindi subito disponibili ma ridondanti e costose da aggiornare;
- **virtuali**: non memorizzate; ogni interrogazione sulla vista viene eseguita sostituendo la vista con la sua definizione. Sono supportate da tutti i DBMS.

Con Supervisione = $\pi_{Impiegato,Capo}$(Afferenza ⋈ Direzione), l'interrogazione $\sigma_{Capo='Leoni'}$(Supervisione) viene eseguita come $\sigma_{Capo='Leoni'}(\pi_{Impiegato,Capo}$(Afferenza ⋈ Direzione)). Le viste mostrano a ogni utente solo ciò che gli interessa e che è autorizzato a vedere, semplificano la scrittura di espressioni complesse e permettono a programmi esistenti di funzionare su schemi ristrutturati; non migliorano però l'efficienza.

Scrivere le interrogazioni: notazione e schemi ricorrenti

Un'interrogazione complessa scritta in una sola riga è quasi illeggibile. Conviene spezzarla in **passi**: ogni passo calcola una relazione intermedia, le dà un nome (R_1, R_2, …) e la usa nei passi successivi, come si fa con le variabili in un programma. Quando la stessa relazione serve due volte (per confrontare un'ennupla con le altre) se ne prendono due **copie** con nomi diversi, e si distinguono gli attributi con il nome della copia davanti: E_1.cachet è il cachet della prima copia, E_2.cachet quello della seconda.

Negli esempi che seguono si usa lo schema Artista(id, nome, nazione), Festival(codice, nome, citta) ed Esibizione(artista, festival, data, cachet), dove artista e festival riferiscono le chiavi delle altre due relazioni. Quasi tutte le interrogazioni ricadono in uno di quattro schemi.

SCHEMA 1: IL MASSIMO O IL MINIMO

L'algebra non sa calcolare un massimo, ma sa **confrontare** ennuple. L'idea è in due tempi: prima si trovano le ennuple che sono "battute" da almeno un'altra con un valore più grande, poi le si tolgono dal totale. Quello che resta sono i massimi. Esempio: le esibizioni con il cachet più alto.

PASSO 1	Si prendono due copie della relazione, per poter confrontare ogni esibizione con tutte le altre. E_1 = Esibizione E_2 = Esibizione
PASSO 2	Si uniscono le copie tenendo solo le coppie in cui la prima esibizione ha un cachet **minore** della seconda. Ogni esibizione che compare a sinistra è quindi battuta da qualcuno: non è un massimo. Si conservano i suoi attributi di identificazione. $B = \pi_{E1.artista,\ E1.festival,\ E1.data}(E_1 \bowtie_{E1.cachet < E2.cachet} E_2)$
PASSO 3	Si tolgono le esibizioni battute da tutte le esibizioni: restano le massime, compresi i pari merito. $Massimi = \pi_{artista,\ festival,\ data}(Esibizione) - B$

Per il **minimo** basta invertire il confronto (E_1.cachet > E_2.cachet). Se il massimo va cercato **per gruppo** (il cachet più alto di ogni festival) si aggiunge al join l'uguaglianza sul gruppo, così si confrontano solo esibizioni dello stesso festival: E_1.festival = E_2.festival ∧ E_1.cachet < E_2.cachet.

SCHEMA 2: "TUTTI"

Le domande con "tutti" si risolvono con la **divisione**. A sinistra si mettono le coppie (soggetto, oggetto) che si sono verificate, a destra l'elenco completo degli oggetti, con **lo stesso nome di attributo**. Esempio: gli artisti che si sono esibiti in tutti i festival.

PASSO 1	Le coppie realizzate: quale artista si è esibito in quale festival. $R_1 = \pi_{\text{artista, festival}}(\text{Esibizione})$
PASSO 2	L'elenco di tutti i festival. In Festival l'attributo si chiama codice, ma per la divisione deve chiamarsi come in R_1: lo si ridenomina in festival. $R_2 = \delta_{\text{codice} \to \text{festival}}(\pi_{\text{codice}}(\text{Festival}))$
PASSO 3	La divisione tiene gli artisti che sono associati a **ogni** festival di R_2. $\text{Risultato} = R_1 \div R_2$

SCHEMA 3: "SOLO", "NESSUNO", "MAI"

Si usa la **differenza**: si parte da un insieme e si tolgono quelli che violano la condizione. Esempio: gli artisti che non si sono mai esibiti.

PASSO 1	Tutti gli artisti. $R_1 = \pi_{\text{id}}(\text{Artista})$
PASSO 2	Gli artisti che si sono esibiti almeno una volta, con l'attributo rinominato in id per poterli sottrarre. $R_2 = \delta_{\text{artista} \to \text{id}}(\pi_{\text{artista}}(\text{Esibizione}))$
PASSO 3	Quelli che restano non si sono mai esibiti. $\text{Risultato} = R_1 - R_2$

Le domande con "solo" seguono la stessa logica: gli artisti che si sono esibiti *solo* a Catania sono quelli che si sono esibiti a Catania, meno quelli che si sono esibiti almeno una volta in un'altra città.

SCHEMA 4: "ALMENO DUE"

Si confronta la relazione con una sua copia cercando due ennuple **diverse** dello stesso soggetto. Esempio: gli artisti che si sono esibiti in almeno due festival diversi.

PASSO 1	Due copie di Esibizione. $E_1 = \text{Esibizione}$ $E_2 = \text{Esibizione}$
PASSO 2	Si cercano coppie con lo stesso artista e festival diversi: se ne esiste almeno una, l'artista ha suonato in almeno due festival. $\text{Risultato} = \pi_{\text{E1.artista}}(E_1 \bowtie_{\text{E1.artista} = \text{E2.artista} \,\wedge\, \text{E1.festival} \neq \text{E2.festival}} E_2)$

METODO: SCRIVERE UN'INTERROGAZIONE IN ALGEBRA

- leggi la domanda e riconosci lo schema: il massimo, tutti, solo, almeno due;
- scrivi un passo per riga, dando un nome a ogni risultato intermedio;
- prima di unione, differenza e divisione controlla che gli attributi abbiano lo stesso nome, ridenominando se serve: è l'errore più frequente;
- ricorda che l'algebra **non ha** conteggi, somme e medie: "il più alto" e "il più basso" diventano confronti fra ennuple.

Esempio svolto: quattro interrogazioni sullo stesso schema

Schema di una cucina professionale: Cuoco(id, nome, stelle), Piatto(codice, nome, categoria) e Prepara(cuoco, piatto), dove Prepara.cuoco riferisce Cuoco e Prepara.piatto riferisce Piatto. Ogni domanda applica uno dei quattro schemi visti sopra.

1. I CUOCHI CHE SANNO PREPARARE TUTTI I DOLCI (SCHEMA "TUTTI")

PASSO 1	Le coppie (cuoco, piatto) realizzate. $R_1 = \pi_{\text{cuoco, piatto}}(\text{Prepara})$
PASSO 2	L'elenco dei dolci, con l'attributo rinominato come in R_1. $D = \delta_{\text{codice} \to \text{piatto}}(\pi_{\text{codice}}(\sigma_{\text{categoria} = \text{'dolce'}}(\text{Piatto})))$
PASSO 3	I cuochi associati a ogni dolce. $\text{Risultato} = R_1 \div D$

2. IL CUOCO CON PIÙ STELLE (SCHEMA "IL MASSIMO")

PASSO 1	Due copie dei cuochi. $C_1 = \text{Cuoco}$ $C_2 = \text{Cuoco}$
PASSO 2	I cuochi battuti da qualcuno con più stelle. $B = \pi_{\text{C1.id}}(C_1 \bowtie_{\text{C1.stelle} < \text{C2.stelle}} C_2)$
PASSO 3	Tutti i cuochi meno quelli battuti. $\text{Risultato} = \pi_{\text{id}}(\text{Cuoco}) - B$

3. I PIATTI PREPARATI DA ALMENO DUE CUOCHI (SCHEMA "ALMENO DUE")

PASSO 1	Due copie di Prepara. P_1 = Prepara P_2 = Prepara
PASSO 2	Coppie con lo stesso piatto e cuochi diversi. $\text{Risultato} = \pi_{P1.\text{piatto}}(P_1 \bowtie_{P1.\text{piatto} = P2.\text{piatto} \wedge P1.\text{cuoco} \neq P2.\text{cuoco}} P_2)$

4. I CUOCHI CHE PREPARANO SOLO PRIMI (SCHEMA "SOLO")

PASSO 1	I piatti che **non** sono primi. $N = \sigma_{\text{categoria} \neq \text{'primo'}}(\text{Piatto})$
PASSO 2	I cuochi che preparano almeno uno di quei piatti: sono quelli da escludere. $A = \pi_{\text{cuoco}}(\text{Prepara} \bowtie_{\text{piatto} = \text{codice}} N)$
PASSO 3	Chi prepara qualcosa, meno chi prepara almeno un piatto che non è un primo. $\text{Risultato} = \pi_{\text{cuoco}}(\text{Prepara}) - A$

Nota sull'ultimo esempio: un cuoco che non prepara nulla non compare nel risultato, perché non sta in $\pi_{\text{cuoco}}(\text{Prepara})$. Se lo si volesse includere si partirebbe da tutti i cuochi, $\pi_{\text{id}}(\text{Cuoco})$, ridenominando id in cuoco.

CAPITOLO 4

Ottimizzazione algebrica delle interrogazioni

Espressioni equivalenti producono lo stesso risultato con costi diversi: le regole di trasformazione e l'algoritmo che anticipa selezioni e proiezioni.

Due espressioni sono **equivalenti** se producono lo stesso risultato qualunque sia l'istanza della base di dati. L'equivalenza è importante in pratica perché il DBMS non esegue l'espressione così come è scritta, ma una equivalente meno costosa. Il costo dipende soprattutto dalla **dimensione dei risultati intermedi**: più presto si riducono le relazioni, meno lavoro fanno i join.

L'esempio classico è il **push della selezione**: se A è un attributo di R_2,

$$\sigma_{A=10}(R_1 \bowtie R_2) = R_1 \bowtie \sigma_{A=10}(R_2)$$

Un esempio concreto: i professori che hanno dato più di 27 a Mario Rossi. Le due espressioni seguenti danno lo stesso risultato, ma la prima costruisce un join enorme e solo alla fine lo filtra, mentre la seconda filtra subito e unisce poche ennuple.

Lenta: $\pi_{\text{Professore}}(\sigma_{\text{Nome = 'Mario Rossi'} \wedge \text{ Voto > 27}}(\text{Studenti} \bowtie \text{Esami} \bowtie \text{Corsi}))$

Veloce: $\pi_{\text{Professore}}(\sigma_{\text{Nome = 'Mario Rossi'}}(\text{Studenti}) \bowtie \sigma_{\text{Voto > 27}}(\text{Esami}) \bowtie \text{Corsi})$

Regole di trasformazione

REGOLA	EQUIVALENZA	CONDIZIONE
1. Raggruppamento di selezioni	$\sigma_{c(X)}(\sigma_{c(Y)}(R)) = \sigma_{c(X) \wedge c(Y)}(R)$	sempre
2a. Commutatività σ e π	$\sigma_{c(X)}(\pi_Y(R)) = \pi_Y(\sigma_{c(X)}(R))$	$X \subseteq Y$
2b. Commutatività σ e π	$\pi_Y(\sigma_{c(X)}(\pi_{XY}(R))) = \pi_Y(\sigma_{c(X)}(R))$	$X \not\subset Y$
3a. Anticipo di σ su ×	$\sigma_{c(X)}(E \times F) = \sigma_{c(X)}(E) \times F$	$X \subseteq \text{attr}(E)$
3b. Anticipo di σ su ×	$\sigma_{c(X) \wedge c(Y)}(E \times F) = \sigma_{c(X)}(E) \times \sigma_{c(Y)}(F)$	$X \subseteq \text{attr}(E)$, $Y \subseteq \text{attr}(F)$
4. Raggruppamento di π	$\pi_X(\pi_Y(E)) = \pi_X(E)$	$X \subseteq Y$
5. π superflue	$\pi_X(E) = E$	$X = \text{attr}(E)$
6. Anticipo di π su ×	$\pi_{XY}(E \times F) = \pi_X(E) \times \pi_Y(F)$	$X \subseteq \text{attr}(E)$, $Y \subseteq \text{attr}(F)$

Valgono inoltre alcune proprietà distributive, utili per spostare selezioni e proiezioni dentro unioni e differenze, e alcune equivalenze per le condizioni composte:

PROPRIETÀ	EQUIVALENZA
σ su unione	$\sigma_C(R_1 \cup R_2) = \sigma_C(R_1) \cup \sigma_C(R_2)$
σ su differenza	$\sigma_C(R_1 - R_2) = \sigma_C(R_1) - \sigma_C(R_2)$
π su unione	$\pi_X(R_1 \cup R_2) = \pi_X(R_1) \cup \pi_X(R_2)$
Condizione in OR	$\sigma_{C \vee D}(R) = \sigma_C(R) \cup \sigma_D(R)$
Condizione in AND	$\sigma_{C \wedge D}(R) = \sigma_C(R) \cap \sigma_D(R)$
Condizione con negazione	$\sigma_{C \wedge \neg D}(R) = \sigma_C(R) - \sigma_D(R)$

ATTENZIONE

La proiezione **non** si distribuisce sulla differenza: $\pi_X(R_1 - R_2)$ è in generale diverso da $\pi_X(R_1) - \pi_X(R_2)$. Controesempio: $R_1 = \{(a, 1)\}$, $R_2 = \{(a, 2)\}$ su (X, Y). A sinistra $\pi_X(\{(a, 1)\}) = \{a\}$; a destra $\{a\} - \{a\} = \emptyset$.

L'algoritmo di ottimizzazione

L'idea è anticipare il più possibile selezioni e proiezioni rispetto ai prodotti (e quindi ai join), che sono le operazioni che fanno crescere i risultati intermedi. Si applicano, finché possibile:

1. si anticipa σ rispetto a π con la regola 2a;
2. si raggruppano le selezioni con la regola 1;
3. si anticipa σ rispetto al prodotto con la regola 3;
4. si eliminano le proiezioni superflue con la regola 5;
5. si raggruppano le proiezioni con la regola 4;
6. si anticipano le proiezioni rispetto al prodotto usando ripetutamente le regole 2b e 6.

I primi tre passi spingono le selezioni verso le foglie dell'albero dell'espressione; gli ultimi tre fanno lo stesso con le proiezioni, eliminando prima possibile le colonne che non servono più. Nei DBMS reali questa **ottimizzazione algebrica** è solo la prima fase: segue l'**ottimizzazione basata sui costi**, che sceglie metodi di accesso e algoritmi di join usando le statistiche del catalogo (capitolo 11).

Esempio svolto: applicare l'algoritmo

Schema: Studenti(Matricola, Nome), Esami(Matricola, Corso, Voto), Corsi(Corso, Professore). Interrogazione: i professori che hanno dato più di 27 a Mario Rossi. Seguiamo l'algoritmo un passo alla volta.

PASSO 1	**Forma iniziale.** È quella che si ottiene traducendo direttamente la query SQL: prodotto di tutte le tabelle, poi una sola grande selezione con quattro condizioni, infine la proiezione. Chiamiamo C la congiunzione delle quattro condizioni: Nome = 'Mario Rossi', Voto > 27, S.Matricola = E.Matricola, E.Corso = C.Corso. $\pi_{\text{Professore}}(\sigma_{C}(\text{Studenti} \times \text{Esami} \times \text{Corsi}))$
PASSO 2	**Regola 1 (passo B).** La selezione con la congiunzione si spezza in selezioni separate, una per condizione, così ognuna si può spostare per conto suo.
PASSO 3	**Regola 3 (passo C).** Ogni selezione scende verso la relazione che contiene i suoi attributi: la condizione sul nome va su Studenti, quella sul voto su Esami. Le due uguaglianze fra attributi restano sopra i prodotti e li trasformano in join. $\sigma_{\text{Nome = 'Mario Rossi'}}(\text{Studenti})$ $\sigma_{\text{Voto > 27}}(\text{Esami})$
PASSO 4	**Regole 4 e 6 (passi E e F).** Si proiettano subito le sole colonne che servono ai passi successivi: la matricola degli studenti, matricola e corso degli esami, corso e professore dei corsi. $S' = \pi_{\text{Matricola}}(\sigma_{\text{Nome = 'Mario Rossi'}}(\text{Studenti}))$ $E' = \pi_{\text{Matricola, Corso}}(\sigma_{\text{Voto > 27}}(\text{Esami}))$
PASSO 5	**Forma ottimizzata.** Si uniscono le relazioni già ridotte e si proietta il risultato. $\pi_{\text{Professore}}(S' \bowtie E' \bowtie \text{Corsi})$

Se Studenti ha 10.000 righe ed Esami 200.000, il prodotto iniziale avrebbe miliardi di combinazioni; dopo le selezioni il primo join coinvolge una sola riga di Studenti e poche decine di esami.

CAPITOLO 5

SQL: le interrogazioni

Il linguaggio dichiarativo standard dei DBMS relazionali: la SELECT, i join, gli operatori insiemistici, l'aggregazione e le interrogazioni nidificate.

SQL (Structured Query Language) nasce in IBM nel 1973 ed è lo standard di tutti i DBMS relazionali. Le versioni principali sono SQL 86, SQL 92 (detto SQL 2, il riferimento di base) e SQL:1999 (SQL 3), che introduce trigger, viste ricorsive e aspetti a oggetti. A differenza dell'algebra, SQL è **dichiarativo**: si descrive che cosa si vuole ottenere, non come calcolarlo. I comandi non distinguono maiuscole e minuscole e possono occupare più righe; per leggibilità ogni clausola va su una riga propria.

CONVENZIONE TIPOGRAFICA

In tutti i blocchi di codice di questa dispensa il segno meno è stampato come −: sulla tastiera si digita con il normale tasto meno. I commenti sono scritti fra /* e */.

La SELECT e la sua semantica

```
SELECT [DISTINCT] ListaAttributiOEspressioni
FROM ListaTabelle
[WHERE Condizione]
[GROUP BY ListaAttributiDiRaggruppamento]
[HAVING CondizioneSuiGruppi]
[ORDER BY ListaAttributiDiOrdinamento]
```

Il significato di una SELECT semplice si legge in tre passi: si fa il **prodotto cartesiano** delle tabelle del FROM, si tengono le righe che soddisfano il **WHERE**, si calcolano le espressioni della **target list**. In algebra:

$$\text{SELECT DISTINCT } A_1, \ldots, A_n \quad \text{FROM } R_1, \ldots, R_m \quad \text{WHERE } C$$

$$\text{corrisponde a} \quad \pi_{A1, \ldots, An}(\sigma_C(R_1 \times \ldots \times R_m))$$

Una differenza fondamentale rispetto all'algebra: in SQL le tabelle sono **multiinsiemi** e i duplicati restano, a meno di chiedere esplicitamente **DISTINCT**. Nella target list si possono usare espressioni aritmetiche (stipendio * 12) e **alias** di colonna con AS (le virgolette servono solo se l'alias contiene spazi); nel FROM si danno alias alle tabelle (FROM Corsi c, Esami e), indispensabili quando la stessa tabella compare più volte o quando un attributo ha lo stesso nome in più tabelle.

PREDICATI DELLA CLAUSOLA WHERE

PREDICATO	SIGNIFICATO
=, <>, <, >, <=, >=	confronti fra espressioni
A BETWEEN x AND y	equivale a x <= A AND A <= y
A IN (v1, v2, …)	A è uno dei valori elencati
A LIKE 'S%'	confronto con un modello: % indica zero o più caratteri, _ esattamente uno
A IS [NOT] NULL	unico modo corretto di controllare i nulli
NOT, AND, OR	connettivi; la precedenza è NOT, poi AND, poi OR; le parentesi la modificano

La precedenza fra AND e OR causa errori frequenti: WHERE job = 'SALESMAN' OR job = 'PRESIDENT' AND sal > 1500 restituisce tutti i venditori e i soli presidenti con stipendio alto, perché AND lega più di OR. Con le parentesi (job = 'SALESMAN' OR job = 'PRESIDENT') AND sal > 1500 si ottiene il significato voluto. **ORDER BY**, sempre ultima, ordina per una o più colonne (ASC predefinito, DESC decrescente), anche non presenti nella target list.

Join

Il join si può scrivere in due modi equivalenti: con la condizione nel WHERE, oppure in modo esplicito nel FROM con **JOIN … ON**, che separa le condizioni di join dalle condizioni di selezione e rende la query più leggibile.

```
/* padre e madre di ogni persona */
SELECT p.figlio, p.padre, m.madre
FROM Paternita p JOIN Maternita m ON p.figlio = m.figlio;

/* persone che guadagnano piu del padre: la tabella Persone compare due volte */
SELECT f.nome, f.reddito, p.reddito AS redditoPadre
FROM Persone p
     JOIN Paternita x ON p.nome = x.padre
     JOIN Persone f ON x.figlio = f.nome
WHERE f.reddito > p.reddito;
```

Le varianti del join esplicito sono: **NATURAL JOIN** (sugli attributi con lo stesso nome), **JOIN … USING (A, B)** (naturale solo sugli attributi elencati), **CROSS JOIN** (prodotto cartesiano) e i join esterni **LEFT**, **RIGHT** e **FULL [OUTER] JOIN**, che conservano le righe senza corrispondenza completandole con NULL. Dimenticare la condizione di join produce un prodotto cartesiano: con 14 impiegati e 4 dipartimenti si ottengono 56 righe prive di senso.

```
/* agenti con l'ammontare dei loro ordini, inclusi quelli senza ordini (ammontare NULL) */
SELECT a.CodiceAgente, o.Ammontare
FROM Agenti a LEFT JOIN Ordini o ON a.CodiceAgente = o.CodiceAgente;
```

Operatori insiemistici

UNION, **INTERSECT** ed **EXCEPT** combinano i risultati di due SELECT con lo stesso numero di colonne di tipo compatibile; eliminano i duplicati, a meno di aggiungere ALL. La corrispondenza fra colonne è **posizionale**: il risultato prende i nomi delle colonne del primo operando, e le ridenominazioni non cambiano l'ordine. Per elencare tutti i genitori con i figli bisogna quindi scrivere SELECT padre AS genitore, figlio FROM Paternita UNION SELECT madre AS genitore, figlio FROM Maternita, con le colonne nello stesso ordine.

Operatori aggregati e raggruppamento

Gli operatori aggregati calcolano un valore a partire da un insieme di righe: **COUNT**, **SUM**, **AVG**, **MAX**, **MIN**. Non hanno equivalente in algebra relazionale. COUNT(*) conta le righe, COUNT(A) conta i valori non nulli di A, COUNT(DISTINCT A) i valori distinti non nulli; SUM, AVG, MAX e MIN ignorano i nulli. Un aggregato si applica al risultato della SELECT dopo il WHERE.

Con **GROUP BY** le righe vengono divise in gruppi con valori uguali sugli attributi di raggruppamento, e ogni gruppo produce una sola riga del risultato. Ne segue la regola d'oro: nella target list possono comparire **solo** attributi di raggruppamento e operatori aggregati. La clausola **HAVING** filtra i gruppi, come il WHERE filtra le righe: le condizioni che contengono aggregati vanno nell'HAVING, tutte le altre nel WHERE.

```
/* per gli studenti con piu di 8 esami: voto minimo, massimo e medio */
SELECT s.Nome, s.Matricola, MIN(e.Voto), MAX(e.Voto), AVG(e.Voto)
FROM Studenti s JOIN Esami e ON s.Matricola = e.Matricola
GROUP BY s.Nome, s.Matricola
HAVING COUNT(*) > 8;
```

L'ordine logico di valutazione è FROM, WHERE, GROUP BY, HAVING, SELECT, ORDER BY. Gli aggregati non si annidano: SELECT AVG(COUNT(*)) non è ammesso, e la media dei conteggi si calcola con una vista o una sottointerrogazione nel FROM che produce prima i conteggi.

Interrogazioni nidificate

Nel WHERE (e nell'HAVING) si può confrontare un valore con il risultato di un'altra SELECT. Le forme sono:

- **A θ (SELECT …)**: la sottointerrogazione deve restituire un solo valore;
- **A θ ANY (SELECT …)**: vero se il confronto vale per almeno un valore restituito;
- **A θ ALL (SELECT …)**: vero se il confronto vale per tutti i valori restituiti;
- **A [NOT] IN (SELECT …)**: IN equivale a = ANY, NOT IN equivale a <> ALL;
- **[NOT] EXISTS (SELECT …)**: vero se la sottointerrogazione restituisce almeno una riga.

```
/* il dipartimento di chi ha lo stipendio piu alto */
SELECT Dipart FROM Impiegato
WHERE Stipendio >= ALL (SELECT Stipendio FROM Impiegato);

/* i dipartimenti la cui somma di stipendi supera la media delle somme */
SELECT Dipart FROM Impiegato
GROUP BY Dipart
HAVING SUM(Stipendio) > (SELECT AVG(Tot)
                         FROM (SELECT SUM(Stipendio) AS Tot
                               FROM Impiegato GROUP BY Dipart) AS T);
```

Una sottointerrogazione è **correlata** quando usa variabili della query esterna: concettualmente viene rieseguita per ogni riga esterna. Le regole di visibilità sono quelle dei linguaggi a blocchi: un blocco interno vede le variabili dei blocchi esterni, ma non viceversa, e due sottointerrogazioni allo stesso livello non si vedono fra loro.

```
/* studenti che hanno un omonimo */
SELECT * FROM Studente s
WHERE EXISTS (SELECT * FROM Studente s2
              WHERE s2.Nome = s.Nome AND s2.Cognome = s.Cognome
                AND s2.Matricola <> s.Matricola);
```

IL QUANTIFICATORE UNIVERSALE CON DOPPIO NOT EXISTS

SQL non ha un "per ogni", ma lo si ottiene da una doppia negazione: "x ha fatto ordini a tutti gli agenti di Catania" equivale a "non esiste un agente di Catania a cui x non abbia fatto un ordine". È la traduzione SQL della divisione:

```
SELECT c.CodiceCliente
FROM Clienti c
WHERE NOT EXISTS (SELECT * FROM Agenti a
                  WHERE a.Zona = 'Catania'
                    AND NOT EXISTS (SELECT * FROM Ordini o
                                    WHERE o.CodiceCliente = c.CodiceCliente
                                      AND o.CodiceAgente = a.CodiceAgente));
```

Un'alternativa spesso più semplice da scrivere usa il conteggio: si raggruppa per cliente e si tiene il gruppo se il numero di agenti distinti di Catania serviti è uguale al numero totale di agenti di Catania.

Valori nulli e logica a tre valori

In SQL un confronto con NULL non è vero né falso ma **unknown**. Il WHERE tiene solo le righe per cui la condizione è **vera**. Le tabelle di verità estese sono: NOT unknown = unknown; vero AND unknown = unknown, falso AND unknown = falso; vero OR unknown = vero, falso OR unknown = unknown. Conseguenze pratiche:

- WHERE anni > 30 esclude le righe con anni nullo, e WHERE anni <= 30 anche: insieme non danno tutta la tabella;
- A <> ALL (sottoquery) è unknown se la sottoquery contiene un NULL: un NOT IN su una colonna con nulli può restituire un risultato vuoto inatteso;
- A θ (sottoquery) vale unknown anche se la sottoquery restituisce una tabella vuota.

METODO: GLI SCHEMI RICORRENTI DELLE QUERY SQL

La maggior parte delle interrogazioni segue pochi schemi ricorrenti:

- "per ogni X, il numero di …": JOIN e GROUP BY X, con COUNT(*) nella target list;
- "maggiore della media": confronto con una sottointerrogazione che calcola AVG; se la media è di conteggi, prima si definisce una vista con i conteggi;
- "il più alto": >= ALL oppure = (SELECT MAX …);
- "tutti": doppio NOT EXISTS, oppure confronto fra COUNT(DISTINCT …) e il totale;
- "i primi tre per gruppo": sottointerrogazione correlata che conta quanti hanno un valore maggiore e tiene le righe con meno di 3 superiori.

Ricorda di mettere nel GROUP BY tutti gli attributi non aggregati della target list.

CAPITOLO 6

SQL: definizione dei dati, viste e controllo dell'accesso

Come si creano schemi, tabelle e domini, come si esprimono i vincoli, come si modificano i dati e come si usano viste, asserzioni e privilegi.

Schemi, tabelle e domini

Uno schema raccoglie gli elementi di una base di dati (tabelle, viste, domini, indici) ed è registrato nel **catalogo** del DBMS. Si crea con CREATE SCHEMA Nome AUTHORIZATION Utente e si elimina con DROP SCHEMA Nome RESTRICT (solo se vuoto) oppure CASCADE (con tutto il contenuto).

CREATE TABLE definisce uno schema di relazione e ne crea un'istanza vuota, specificando attributi, domini e vincoli. I domini predefiniti sono le stringhe CHAR(n) a lunghezza fissa e VARCHAR(n) variabile, i numerici esatti INTEGER, SMALLINT, NUMERIC(p, s) con p cifre di cui s decimali, gli approssimati REAL e FLOAT, le date e gli orari DATE e TIME. CREATE DOMAIN definisce domini riutilizzabili con vincoli e valori predefiniti:

```
CREATE DOMAIN Voto AS SMALLINT DEFAULT NULL
    CHECK (VALUE >= 18 AND VALUE <= 30);
```

Vincoli intrarelazionali

- **NOT NULL**: il valore è obbligatorio (implicito per la chiave primaria);
- **UNIQUE**: l'attributo, o l'insieme di attributi, è una chiave;
- **PRIMARY KEY**: la chiave primaria, una sola per tabella, senza nulli;
- **CHECK (condizione)**: vincolo di ennupla su uno o più attributi della stessa riga;
- **DEFAULT valore**: valore usato quando l'inserimento non lo specifica.

Un vincolo su un solo attributo si può scrivere accanto all'attributo; un vincolo su più attributi va scritto come elemento separato. Attenzione alla differenza: UNIQUE (Cognome, Nome) dice che la *coppia* non si ripete, mentre Nome UNIQUE e Cognome UNIQUE dicono che non si ripetono né i nomi né i cognomi, un vincolo molto più forte.

```
CREATE TABLE Esami (
    Studente CHAR(6),
    Corso    INTEGER,
    Voto     INTEGER NOT NULL CHECK (Voto >= 18 AND Voto <= 30),
    Lode     INTEGER NOT NULL DEFAULT 0,
    CHECK (Lode = 0 OR (Voto = 30 AND Lode = 1)),   /* lode solo con 30 */
    PRIMARY KEY (Studente, Corso),
    FOREIGN KEY (Studente) REFERENCES Studenti(Matricola),
    FOREIGN KEY (Corso) REFERENCES Corsi(Codice)
);
```

Integrità referenziale e politiche di reazione

FOREIGN KEY (attributi) REFERENCES Tabella(attributi) definisce una chiave esterna; per un solo attributo si può scrivere REFERENCES accanto alla colonna. Il DBMS rifiuta gli inserimenti nella tabella che riferisce con valori che non esistono nella tabella riferita. Per le modifiche alla tabella riferita si sceglie una politica:

POLITICA	EFFETTO SU ON DELETE (ANALOGO PER ON UPDATE)
NO ACTION / RESTRICT	l'operazione viene rifiutata se esistono righe che riferiscono quella cancellata (predefinita)
CASCADE	si cancellano anche le righe che la riferiscono (con ON UPDATE si propaga il nuovo valore)
SET NULL	la chiave esterna delle righe che la riferiscono diventa NULL
SET DEFAULT	la chiave esterna prende il valore predefinito

Modifica dei dati e degli schemi

```
INSERT INTO Esami (Studente, Corso, Voto) VALUES ('123456', 3, 27);
INSERT INTO EsamiBuoni SELECT * FROM Esami WHERE Voto > 27;
DELETE FROM Esami WHERE Studente = '123456';
UPDATE Impiegato SET Stipendio = Stipendio * 1.1 WHERE Dipart = 'Vendite';
```

INSERT accetta una lista di valori oppure il risultato di una SELECT; DELETE e UPDATE agiscono su tutte le righe che soddisfano il WHERE (senza WHERE, su tutta la tabella). Gli schemi si modificano con ALTER TABLE (aggiungere o eliminare colonne e vincoli) e si eliminano con DROP TABLE, DROP DOMAIN, DROP VIEW, con le opzioni RESTRICT e CASCADE. CREATE TABLE Nuova AS SELECT … crea una tabella popolata dal risultato di una query.

Viste in SQL

```
CREATE VIEW MediaVoti (Matricola, Media) AS
    SELECT Matricola, AVG(Voto) FROM Esami GROUP BY Matricola;
```

Le viste servono a semplificare query complesse (per esempio la media dei conteggi, impossibile con un solo livello di aggregazione), a proteggere i dati (una vista senza la colonna Saldo concessa al posto della tabella), a scomporre problemi in passi e a riorganizzare gli schemi mantenendo la compatibilità con i programmi esistenti. Una vista si usa come una tabella e si può definire su altre viste.

VISTE AGGIORNABILI

Aggiornare una vista significa aggiornare le tabelle di base, e ciò è possibile solo se ogni riga e colonna della vista corrisponde a una e una sola riga e colonna di una tabella di base. In pratica sono aggiornabili le viste su una sola tabella che ne contengono la chiave e non usano aggregati, DISTINCT o raggruppamenti; le viste su più tabelle in genere non lo sono. L'opzione **WITH CHECK OPTION** impedisce inserimenti e modifiche che produrrebbero righe non visibili attraverso la vista: su una vista con WHERE Cognome = 'Rossi', inserire un 'Bianchi' viene rifiutato.

VISTE RICORSIVE

Alcune interrogazioni, come la **chiusura transitiva** (tutti i superiori di un impiegato, a qualsiasi livello), non sono esprimibili né in algebra né in SQL 92, perché servirebbe un numero di join che dipende dai dati. SQL:1999 introduce le viste ricorsive:

```
WITH RECURSIVE Discendenza (Antenato, Discendente) AS (
    SELECT Padre, Figlio FROM Paternita            /* passo base */
    UNION ALL
    SELECT d.Antenato, p.Figlio                     /* passo ricorsivo */
    FROM Discendenza d JOIN Paternita p ON d.Discendente = p.Padre
)
SELECT * FROM Discendenza;
```

Vincoli generici: CHECK e asserzioni

La clausola CHECK può contenere anche sottointerrogazioni, ed esprimere così vincoli che coinvolgono altre righe. Quando un vincolo non appartiene a nessuna tabella in particolare si usa un'**asserzione**, un vincolo a livello di schema. Il modo tipico di scriverla è "non esiste una situazione che violi la regola":

```
CREATE ASSERTION ControlloSalari CHECK (
  NOT EXISTS (SELECT *
              FROM Impiegato i JOIN Dipartimento d ON i.Dipartimento = d.Nome
              WHERE d.Citta = 'Firenze'
                AND i.Stipendio > (SELECT MIN(Stipendio) FROM Impiegato
                                   WHERE Dipartimento = 'Direzione')));
```

NOTA PRATICA

Le asserzioni fanno parte dello standard ma quasi nessun DBMS le implementa, MySQL compreso, e MySQL non accetta sottointerrogazioni nei CHECK. Nei sistemi reali questi vincoli si realizzano con i **trigger** (capitolo 7), come si vede nella dispensa di laboratorio.

Funzioni scalari

Lavorano riga per riga e restituiscono un valore: funzioni temporali (CURRENT_DATE, EXTRACT(YEAR FROM data)), sulle stringhe (CHAR_LENGTH, LOWER), di conversione (CAST(x AS tipo)) e condizionali. COALESCE(a, b, …) restituisce il primo argomento non nullo, NULLIF(a, b) restituisce NULL se a = b, CASE realizza una scelta:

```
SELECT Targa,
       CASE Tipo WHEN 'Auto' THEN 2.58 * KWatt
                 WHEN 'Moto' THEN 22.00 + 1.00 * KWatt
                 ELSE NULL END AS Tassa
FROM Veicolo;
```

Controllo dell'accesso

Ogni risorsa (tabella, vista, attributo, dominio) appartiene a chi la crea, che ha su di essa tutti i privilegi; l'amministratore li ha su tutto. Un privilegio è caratterizzato da risorsa, utente che lo concede, utente che lo riceve, azione permessa e trasmissibilità. Le azioni sono INSERT, UPDATE, DELETE, SELECT, REFERENCES (definire chiavi esterne verso la risorsa) e USAGE (usare un dominio in una definizione).

```
GRANT SELECT, INSERT ON Dipartimento TO Stefano WITH GRANT OPTION;
REVOKE INSERT ON Dipartimento FROM Stefano CASCADE;
```

WITH GRANT OPTION permette al destinatario di concedere a sua volta il privilegio; REVOKE con CASCADE revoca anche i privilegi concessi a catena. Un buon sistema di autorizzazioni non deve rivelare l'esistenza delle

risorse protette: chi non è autorizzato riceve lo stesso messaggio di chi chiede una tabella inesistente. Per autorizzare l'accesso solo ad alcune righe si definisce una **vista** con la selezione e si concede il privilegio sulla vista. SQL:1999 introduce i **ruoli**, insiemi di privilegi assegnati agli utenti.

METODO: VINCOLI CON CHECK E ASSERZIONI

Quando un vincolo coinvolge più righe o più tabelle, la forma più sicura è l'asserzione con NOT EXISTS: si descrive la situazione proibita e se ne nega l'esistenza. Esempio, in un cinema con Sala(numero, posti), Proiezione(id, film, sala, data_ora) e Biglietto(id, proiezione, posto): per nessuna proiezione si possono vendere più biglietti dei posti della sala.

```
CREATE ASSERTION PostiDisponibili CHECK (
  NOT EXISTS (SELECT *
              FROM Proiezione p JOIN Sala s ON p.sala = s.numero
              WHERE s.posti < (SELECT COUNT(*)            /* biglietti venduti */
                               FROM Biglietto b
                               WHERE b.proiezione = p.id)));
```

La lettura è: non esiste una proiezione la cui sala ha meno posti dei biglietti venduti. Lo stesso schema vale per i vincoli di conteggio ("al massimo N", "meno della media"): si raggruppa, si confronta con HAVING o con una sottointerrogazione e si nega con NOT EXISTS. In MySQL, che non ha asserzioni, lo si trasforma in un trigger BEFORE INSERT che esegue lo stesso conteggio.

CAPITOLO 7

Basi di dati attive: i trigger

Regole che il DBMS esegue automaticamente al verificarsi di eventi: il paradigma evento, condizione, azione, la sintassi di SQL:1999 e le proprietà di terminazione.

Una base di dati **attiva** contiene regole, i **trigger**, che reagiscono agli eventi senza che l'applicazione debba intervenire. Seguono il paradigma **evento, condizione, azione** (ECA): quando si verifica l'**evento** (una INSERT, DELETE o UPDATE su una tabella) il trigger è *attivato*; viene poi *considerato*, cioè ne viene valutata la **condizione**; se è vera viene *eseguita* l'**azione**, una sequenza di comandi SQL o una procedura. I trigger servono a mantenere vincoli complessi, dati derivati e ridondanti, registri di controllo e regole aziendali.

Sintassi SQL:1999

```
CREATE TRIGGER NomeTrigger
{ BEFORE | AFTER }
{ INSERT | DELETE | UPDATE [OF Colonna] } ON Tabella
[REFERENCING { [OLD TABLE [AS] VecchiaTab] [NEW TABLE [AS] NuovaTab] }
           | { [OLD [ROW] [AS] VecchiaRiga] [NEW [ROW] [AS] NuovaRiga] }]
[FOR EACH { ROW | STATEMENT }]
[WHEN Condizione]
ComandiSQL
```

Ogni trigger ha un nome, una tabella **target**, una **modalità** (before o after), un **evento**, una **granularità**, eventuali alias per i valori di transizione, una condizione, un'azione e un timestamp di creazione.

MODALITÀ BEFORE E AFTER

Un trigger **BEFORE** viene considerato prima che la modifica sia applicata: non può modificare lo stato della base di dati, ma può controllare la modifica ed eventualmente "correggerla" assegnando i valori NEW, quindi agisce da **condizionatore**. Un trigger **AFTER**, il più comune, agisce dopo la modifica e può eseguire qualsiasi comando, quindi agisce da **reinstallatore**. Lo stesso vincolo, aumenti di stipendio al massimo del 20 per cento, nelle due forme:

```
CREATE TRIGGER LimitaAumenti                    /* condizionatore */
BEFORE UPDATE OF Salario ON Impiegato
FOR EACH ROW
WHEN (NEW.Salario > OLD.Salario * 1.2)
SET NEW.Salario = OLD.Salario * 1.2;

CREATE TRIGGER LimitaAumenti                    /* reinstallatore */
AFTER UPDATE OF Salario ON Impiegato
FOR EACH ROW
WHEN (NEW.Salario > OLD.Salario * 1.2)
UPDATE Impiegato SET Salario = OLD.Salario * 1.2
WHERE Matricola = NEW.Matricola;
```

GRANULARITÀ E VARIABILI DI TRANSIZIONE

In modalità **row level** (FOR EACH ROW) il trigger è considerato una volta per ogni riga modificata e dispone delle variabili OLD e NEW, la riga prima e dopo la modifica. In modalità **statement level** (FOR EACH STATEMENT, predefinita nello standard) è considerato una volta per comando, indipendentemente dal numero di righe, e dispone delle tabelle di transizione OLD TABLE e NEW TABLE. OLD e OLD TABLE non esistono per l'evento INSERT; NEW e NEW TABLE non esistono per DELETE. Nel primo esempio seguente ogni aumento di saldo viene registrato con il saldo precedente e quello nuovo.

```
CREATE TRIGGER MonitoraConti
AFTER UPDATE ON Conto
FOR EACH ROW
WHEN (NEW.Totale > OLD.Totale)
INSERT INTO Versamenti VALUES (NEW.NumConto, OLD.Totale, NEW.Totale);

CREATE TRIGGER ArchiviaFatture                    /* statement level */
AFTER DELETE ON Fattura
REFERENCING OLD TABLE AS FattureCancellate
INSERT INTO FattureArchiviate (SELECT * FROM FattureCancellate);
```

Esecuzione e conflitti fra trigger

Se più trigger sono associati allo stesso evento, in SQL:1999 vengono eseguiti prima i trigger BEFORE (statement e row level), poi la modifica con la verifica dei vincoli di integrità, infine i trigger AFTER (row e statement level). A parità di categoria conta il timestamp di creazione: i trigger più vecchi hanno priorità.

L'azione di un trigger può generare eventi che attivano altri trigger. Ogni attivazione viene eseguita in un nuovo **contesto di esecuzione** (TEC): lo stato del contesto esterno viene salvato e ripristinato al termine di quello interno, in modo **ricorsivo**. L'esecuzione termina correttamente in uno **stato quiescente**, oppure con un errore se si supera una profondità massima di ricorsione; un errore in qualunque punto della catena annulla il comando che l'ha avviata.

In Oracle la sintassi è diversa: ammette più eventi per trigger, non ha tabelle di transizione, la condizione WHEN esiste solo a livello di riga e l'azione è un blocco PL/SQL; l'ordine è BEFORE statement, BEFORE row, modifica, AFTER row, AFTER statement. Se una catena attivata da un trigger cerca di modificare la tabella target, si ottiene la *mutating table exception*. In MySQL esistono solo trigger row level (dettagli in laboratorio).

Proprietà formali e terminazione

- **terminazione**: per ogni stato iniziale e ogni transazione si raggiunge uno stato quiescente;
- **confluenza**: lo stato finale è unico, indipendentemente dall'ordine di esecuzione dei trigger;
- **univoca osservabilità**: i trigger sono confluenti e producono verso l'esterno gli stessi effetti.

La terminazione, la proprietà più importante, si analizza con il **grafo di triggering**: un nodo per trigger e un arco da t_i a t_j se l'azione di t_i può attivare t_j. Se il grafo è aciclico l'esecuzione termina sicuramente. Se ha cicli può non terminare, e bisogna esaminarli uno per uno: un trigger che a ogni esecuzione riduce lo stipendio del 10 per cento finché resta sopra una soglia termina, mentre uno che lo moltiplica per un fattore maggiore di 1 entra in un ciclo infinito.

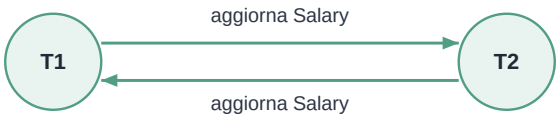


Figura 7.1. Grafo di triggering con un ciclo: T1 aggiorna i contributi quando cambia lo stipendio, T2 riduce gli stipendi oltre una soglia. Il ciclo va analizzato per capire se il sistema termina.

METODO: PROGETTARE UN TRIGGER

Quasi tutti i trigger applicativi **impediscono** un'operazione o **mantiene** un dato. Lo schema da seguire:

- scegli l'evento: quale INSERT, UPDATE o DELETE può violare la regola? Spesso è l'inserimento nella tabella "di dettaglio" (una nuova esibizione, un nuovo biglietto, una nuova visita);
- per impedire usa BEFORE e FOR EACH ROW, calcola i valori di confronto con SELECT … INTO su variabili e, se la regola è violata, solleva un errore (in MySQL SIGNAL SQLSTATE '45000');
- per mantenere un attributo ridondante usa AFTER e un UPDATE sulla tabella che contiene il dato derivato.

Esempio: impedire l'inserimento di un'esibizione con data successiva alla data_fine del festival. Evento BEFORE INSERT ON Esibizione; condizione NEW.data > (SELECT data_fine FROM Festival WHERE codice = NEW.festival); azione: errore. Il codice completo è nel capitolo 15.

Altri esempi di trigger

MANTENERE UN ATTRIBUTO DERIVATO

In una piattaforma di raccolta fondi con Campagna(id, obiettivo, raggiunta) e Donazione(id, campagna, importo): quando la somma delle donazioni di una campagna raggiunge l'obiettivo, il campo raggiunta diventa vero. È un trigger AFTER, perché agisce su un'altra tabella dopo l'inserimento della donazione.

```
CREATE TRIGGER ObiettivoRaggiunto
AFTER INSERT ON Donazione
FOR EACH ROW
WHEN ((SELECT SUM(importo) FROM Donazione WHERE campagna = NEW.campagna)
      >= (SELECT obiettivo FROM Campagna WHERE id = NEW.campagna))
UPDATE Campagna SET raggiunta = TRUE WHERE id = NEW.campagna;
```

AGGIORNARE UN CONTATORE RIDONDANTE

Se si decide di mantenere biglietti_venduti in Festival (analisi del capitolo 15), servono due trigger, uno per l'inserimento e uno per la cancellazione, altrimenti il dato ridondante diventa incoerente.

```
CREATE TRIGGER ContaBiglietti
AFTER INSERT ON Biglietto
FOR EACH ROW
UPDATE Festival SET biglietti_venduti = biglietti_venduti + 1
WHERE codice = NEW.festival;

CREATE TRIGGER ScontaBiglietti
AFTER DELETE ON Biglietto
FOR EACH ROW
UPDATE Festival SET biglietti_venduti = biglietti_venduti - 1
WHERE codice = OLD.festival;
```

CAPITOLO 8

Progettazione concettuale e modello ER

Dalla raccolta dei requisiti allo schema concettuale: i costrutti del modello Entità Relazione, le cardinalità, gli identificatori, le generalizzazioni e le strategie di progetto.

Il ciclo di vita e le tre fasi della progettazione

La realizzazione di una base di dati segue il ciclo di vita dei sistemi informativi: raccolta e analisi dei requisiti, progettazione, implementazione, validazione e collaudo, funzionamento e manutenzione. La progettazione, per il principio ingegneristico di separare il *che cosa* dal *come*, si divide in tre fasi:

FASE	INGRESSO	USCITA
Concettuale	requisiti in linguaggio naturale, moduli, documenti	schema concettuale (ER), indipendente dal DBMS
Logica	schema concettuale e carico applicativo	schema logico (relazionale) ottimizzato e normalizzato
Fisica	schema logico e caratteristiche del DBMS	schema fisico: strutture di memorizzazione e indici

Raccolta e analisi dei requisiti

L'analisi individua proprietà e funzionalità del sistema, descrive i dati e le operazioni e produce uno studio di fattibilità con costi, benefici e piano di sviluppo. Per ogni settore si intervistano i responsabili e si scrive una prima versione dei requisiti in linguaggio naturale, che va ripulita dalle ambiguità tipiche:

- **omonimi**: lo stesso termine per concetti diversi (titolo di un libro e titolo di studio di una persona);
- **sinonimi**: termini diversi per lo stesso concetto (cliente e acquirente, aula e locale);
- termini troppo generici o troppo specifici, frasi contorte, riferimenti impliciti fra termini.

Le regole pratiche sono: scegliere il giusto livello di astrazione, standardizzare la struttura delle frasi, unificare sinonimi e omonimi, rendere espliciti i riferimenti. Il risultato è il **glossario dei termini**, che per ogni termine riporta descrizione, sinonimi e termini collegati, e l'elenco delle **operazioni**.

TERMINE	DESCRIZIONE	SINONIMI	COLLEGAMENTI
Cliente	Persona o azienda che ordina prodotti	Acquirente	Prodotto
Prodotto	Articolo venduto, con codice e genere	Articolo, voce	Fornitore, cliente, scaffale
Scaffale	Supporto numerato del magazzino	Ripiano	Prodotto, operaio

I costrutti del modello Entità Relazione

Il modello ER descrive la realtà con costrutti grafici semplici e indipendenti dall'organizzazione dei dati nel computer. Si distingue fra **conoscenza astratta** (entità, associazioni, attributi) e **conoscenza concreta** (le loro istanze).

- **Entità**: classe di oggetti con proprietà comuni e esistenza autonoma (Studente, Corso, Volo); un'istanza è un oggetto specifico (lo studente 567345). Si disegna con un rettangolo.

- **Associazione** (relationship): legame logico fra due o più entità (Esame lega Studente e Corso); un'istanza è una combinazione di istanze delle entità coinvolte. Si disegna con un rombo.
- **Attributo**: proprietà elementare di un'entità o di un'associazione, con valori in un dominio (Voto di Esame). Si disegna con un pallino collegato al costrutto.

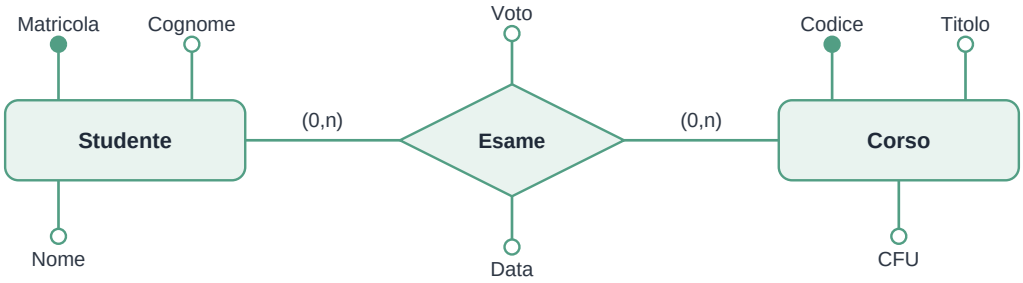


Figura 8.1. Uno schema ER minimo: le entità Studente e Corso, l'associazione Esame con i suoi attributi e le cardinalità. I pallini pieni indicano gli identificatori.

Un'associazione può essere **ricorsiva** (o ad anello) se lega un'entità a se stessa, come Matrimonio su Impiegato o Controllo fra impiegato controllore e controllato; in questo caso si indicano i **ruoli** sui rami. Un'associazione può essere anche **ternaria** o di grado superiore, come Guida fra Guidatore, TIR e Percorso. Attenzione: un'istanza di associazione è identificata dalle istanze che collega, quindi la stessa coppia studente e corso può comparire una sola volta in Esame. Se lo stesso fatto può ripetersi (più forniture dello stesso prodotto dallo stesso fornitore in date diverse) l'associazione va trasformata in un'entità.

TIPI DI ATTRIBUTI

- **semplice** o **composto**: un indirizzo composto da via, civico e CAP;
- **monovalore** o **multivalore**: i titoli di studio di una persona, con cardinalità (1,n);
- **obbligatorio** (1,1) o **opzionale** (0,1): il telefono può mancare;
- **derivato**: calcolabile da altri dati (età da data di nascita, numero di abitanti contando i residenti); la regola di calcolo si documenta a parte.

Cardinalità

Per ogni entità che partecipa a un'associazione si indica una coppia (min, max): il numero minimo e massimo di istanze dell'associazione a cui ogni istanza dell'entità può partecipare. Il minimo è 0 (partecipazione **opzionale**) o 1 (**obbligatoria**); il massimo è 1 o n. Guardando i massimi si classificano le associazioni binarie:

TIPO	MASSIMI	ESEMPIO
Uno a uno	1 e 1	Direzione fra Reparto (1,1) e Direttore (1,1)
Uno a molti	1 da un lato, n dall'altro	Residenza: Persona (1,1), Città (1,n)
Molti a molti	n e n	Esame: Studente (0,n), Corso (0,n)

Le cardinalità si leggono "dal lato dell'entità": in Residenza, il (1,1) accanto a Persona significa che ogni persona risiede in una e una sola città; l'(1,n) accanto a Città che ogni città ha almeno un residente. Anche gli attributi hanno cardinalità, omessa quando è (1,1).

Identificatori

Un **identificatore** (chiave) è un insieme di attributi e di entità che individua univocamente ogni istanza di un'entità. È **interno** se formato solo da attributi dell'entità (la Matricola), **esterno** se coinvolge altre entità collegate: un corso di laurea è identificato dal proprio nome *e* dal dipartimento che lo eroga. Regole: l'identificazione esterna passa solo attraverso associazioni binarie in cui l'entità da identificare partecipa con cardinalità (1,1); può coinvolgere entità a loro volta identificate esternamente, purché non si creino cicli. Ogni entità deve avere almeno un identificatore; le associazioni non hanno identificatori propri, perché sono identificate dalle entità che collegano.

La scelta della chiave dipende anche da fattori pratici: la matricola è comoda da digitare, il codice fiscale è controllabile; spesso si introduce un codice progressivo generato dal sistema (AUTO_INCREMENT).

Generalizzazioni

Una **generalizzazione** lega un'entità padre E a una o più entità figlie E_1, …, E_n che ne sono casi particolari: ogni istanza di una figlia è anche istanza del padre (relazione ISA, "è un"). Le figlie **ereditano** attributi, associazioni e identificatori del padre, che quindi non vanno ripetuti. Se la figlia è una sola si parla di **sottoinsieme**.

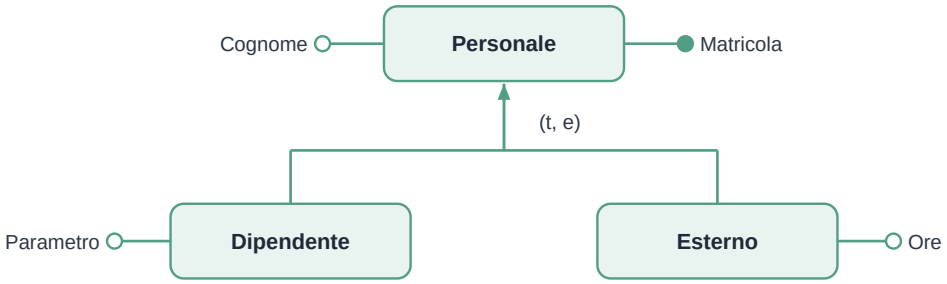


Figura 8.2. Generalizzazione totale ed esclusiva: ogni membro del personale è un dipendente oppure un esterno, mai entrambi; le figlie ereditano Matricola e Cognome.

Le generalizzazioni si classificano con due proprietà, annotate accanto alla freccia:

- **totale** (t) se ogni istanza del padre appartiene ad almeno una figlia, **parziale** (p, o nt) altrimenti: i cittadini cacciatori o pescatori sono un sottoinsieme dei cittadini;
- **esclusiva** (e) se ogni istanza del padre appartiene al più a una figlia, **sovrapposta** (o, o ne) altrimenti: un cittadino può essere sia cacciatore sia pescatore.

Documentazione dello schema

Lo schema ER va corredato di documentazione, perché alcune proprietà non sono esprimibili graficamente. Il **dizionario dei dati** comprende una tabella delle entità (nome, descrizione, attributi, identificatore) e una delle associazioni (nome, descrizione, entità coinvolte con cardinalità, attributi). Le **regole aziendali** (business rules) completano lo schema:

- **vincoli di integrità**, nella forma "concetto deve o non deve espressione": lo stipendio di un impiegato non deve superare quello del direttore del suo dipartimento;
- **regole di derivazione**, nella forma "concetto si ottiene con operazione": il numero di impiegati di un dipartimento si ottiene contando gli impiegati che vi afferiscono.

Strategie di progetto

STRATEGIA	COME PROCEDE	PRO E CONTRO
Top down	da uno schema iniziale astratto, con raffinamenti successivi (primitive: un'entità diventa più entità o una generalizzazione, un'associazione diventa più associazioni o un'entità, si aggiungono attributi)	visione d'insieme subito; difficile se all'inizio non si conosce tutto il sistema
Bottom up	si progettano schemi separati per le parti della specifica e poi si integrano	adatta al lavoro di gruppo; l'integrazione di schemi diversi è delicata
Inside out	si parte da alcuni concetti importanti e si estende "a macchia d'olio"	variante della bottom up, procede per vicinanza
Mista	si divide la specifica in parti, ma si fissa subito uno schema scheletro con i concetti principali	la più usata: unisce la visione d'insieme alla divisione del lavoro

La **metodologia generale** combina questi elementi: analisi dei requisiti con glossario; passo base con lo schema scheletro; eventuale decomposizione dei requisiti; passo iterativo in cui si raffinano i concetti e se ne aggiungono di nuovi finché tutte le specifiche sono rappresentate; integrazione dei sottoschemi; analisi di qualità.

Qualità di uno schema concettuale

- **correttezza**: i costrutti sono usati in modo sintatticamente e semanticamente corretto (non una generalizzazione fra associazioni, non un'associazione al posto di una generalizzazione);
- **completezza**: tutti i dati di interesse sono rappresentati e tutte le operazioni sono eseguibili;
- **leggibilità**: costrutti con più legami al centro, linee perpendicolari con pochi incroci, padri sopra i figli;
- **minimalità**: ogni specifica è rappresentata una sola volta; le ridondanze, se presenti, devono essere una scelta consapevole e documentata.

METODO: DISEGNARE UNO SCHEMA ER

I casi sono due: progettare uno schema da un testo di requisiti, oppure ricostruire lo schema ER da uno schema relazionale dato. Nel secondo caso si procede all'inverso della traduzione:

- ogni relazione con una chiave propria diventa un'**entità**;
- ogni chiave esterna diventa un'**associazione uno a molti**, con (1,1) dal lato della relazione che la contiene;
- ogni relazione la cui chiave è formata da due chiavi esterne diventa un'**associazione molti a molti**, con i suoi attributi;
- alla fine si aggiungono **cardinalità e identificatori**, che sono parte integrante dello schema.

CAPITOLO 9

Progettazione logica

Dallo schema ER allo schema relazionale: analisi delle prestazioni, decisione sulle ridondanze, eliminazione delle generalizzazioni e regole di traduzione.

La progettazione logica riceve lo schema ER, il **carico applicativo** previsto e il modello logico di destinazione, e produce lo schema logico con i vincoli e la documentazione. Si svolge in due fasi: la **ristrutturazione** dello schema ER, indipendente dal modello logico e guidata dalle prestazioni, e la **traduzione** verso il modello relazionale.

Analisi delle prestazioni

Le prestazioni di uno schema ER si stimano con due indici: il **costo di un'operazione**, misurato come numero medio di occorrenze di entità e associazioni visitate per eseguirla, e l'**occupazione di memoria**, in byte. Servono quindi due tavole:

- la **tavola dei volumi**: per ogni entità e associazione (tipo E o R) il numero medio di occorrenze; i volumi delle associazioni si ricavano da quelli delle entità e dalle cardinalità medie;
- la **tavola delle operazioni**: per ogni operazione il tipo (I interattiva, B batch) e la frequenza.

Per ogni operazione si disegna lo **schema di navigazione** (la parte di schema attraversata) e si compila la **tavola degli accessi**, con una riga per ogni costrutto visitato: nome, tipo (E o R), numero di accessi, tipo di accesso, L lettura o S scrittura. Per convenzione **una scrittura costa come due letture** (1S = 2L).

Analisi delle ridondanze

Una **ridondanza** è un'informazione significativa ma derivabile da altre. Le forme tipiche sono gli attributi derivabili da altri attributi della stessa entità (importo lordo da netto e IVA), da attributi di altre entità (importo totale di un acquisto dai prezzi dei prodotti), da operazioni di conteggio (numero di abitanti di una città contando i residenti) e le associazioni derivabili dalla composizione di altre in presenza di cicli. La ridondanza **riduce gli accessi** per leggere il dato derivato, ma **occupa memoria** e richiede **operazioni aggiuntive** per mantenerlo aggiornato. Si decide confrontando i costi con e senza.

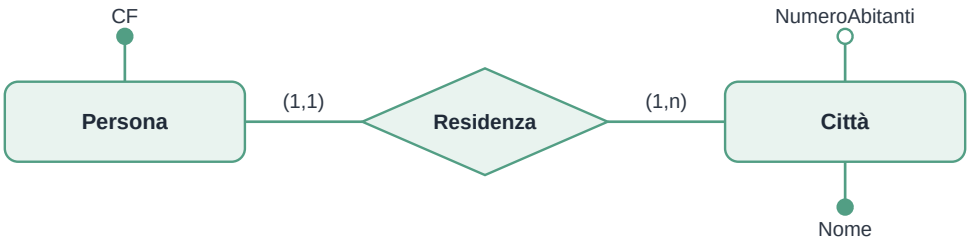


Figura 9.1. L'attributo NumeroAbitanti di Città è ridondante: si può ricavare contando le occorrenze di Residenza.

ESEMPIO SVOLTO: NUMERO DI ABITANTI

Volumi: 200 città e 1.000.000 di persone, quindi 1.000.000 di occorrenze di Residenza. Operazioni: O1, memorizzare una nuova persona con la sua città, 500 volte al giorno; O2, stampare tutti i dati di una città compreso il numero di abitanti, 2 volte al giorno. Si conta 1 scrittura (S) come 2 letture (L).

PASSO 1	**Numero medio di residenti per città.** Serve per sapere quante occorrenze di Residenza bisogna leggere per contare gli abitanti di una città. 1.000.000 / 200 = 5.000 residenti per città
PASSO 2	**Con la ridondanza, operazione O1.** Si scrive la nuova persona, si scrive la residenza, si legge la città e la si riscrive con il contatore aumentato di uno. Persona 1S + Residenza 1S + Città 1L + Città 1S = 3S + 1L = 7L 7L × 500 al giorno = 3.500 letture
PASSO 3	**Con la ridondanza, operazione O2.** Il numero di abitanti è già scritto nella città: basta leggerla. Città 1L × 2 al giorno = 2 letture
PASSO 4	**Senza la ridondanza, operazione O1.** Non c'è nessun contatore da aggiornare. Persona 1S + Residenza 1S = 2S = 4L 4L × 500 al giorno = 2.000 letture
PASSO 5	**Senza la ridondanza, operazione O2.** Per contare gli abitanti si leggono tutte le residenze della città. Città 1L + Residenza 5.000L = 5.001L 5.001L × 2 al giorno = 10.002 letture
PASSO 6	**Confronto.** Si sommano le due operazioni in ciascun caso. con ridondanza: 3.500 + 2 = 3.502 letture al giorno senza ridondanza: 2.000 + 10.002 = 12.002 letture al giorno

La ridondanza costa 4 byte per città, cioè 800 byte in tutto, e fa risparmiare circa 8.500 accessi al giorno: conviene mantenerla.

METODO: CONVIENE MANTENERE L'ATTRIBUTO RIDONDANTE?

Il procedimento per decidere, in ordine:

- ricava dai volumi il **numero medio** di occorrenze per elemento (biglietti per festival, visite per animale);
- **con ridondanza**: l'operazione di inserimento costa la scrittura del nuovo elemento più una lettura e una scrittura dell'attributo ridondante; la lettura del dato derivato costa una sola lettura;
- **senza ridondanza**: l'inserimento costa solo la scrittura del nuovo elemento; la lettura del dato richiede di leggere tutte le occorrenze collegate (il numero medio calcolato al primo punto);
- converti con 1S = 2L, moltiplica per le frequenze, somma e confronta; motiva la conclusione con i numeri.

Se un volume necessario non è dato (per esempio il numero totale delle occorrenze collegate), dichiara un'ipotesi esplicita oppure calcola la soglia oltre la quale la ridondanza conviene.

Eliminazione delle generalizzazioni

Il modello relazionale non rappresenta le generalizzazioni, che vanno sostituite con entità e associazioni. Le alternative sono tre:

SOLUZIONE	COME	QUANDO CONVIENE
Collasso verso l'alto	si eliminano le figlie e il padre ne riceve tutti gli attributi (resi opzionali) più un attributo **selettore** che indica il tipo	operazioni che leggono insieme padre e figlie; poche differenze fra le figlie; produce valori nulli
Collasso verso il basso	si elimina il padre e ogni figlia ne eredita attributi e associazioni (replicate)	molti attributi specifici e accessi separati alle figlie; possibile solo se la generalizzazione è totale; se non è esclusiva introduce ridondanza
Sostituzione con associazioni	si mantengono tutte le entità; ogni figlia è legata al padre da un'associazione uno a uno e identificata esternamente da esso	sempre possibile; conviene quando le operazioni distinguono le figlie dal padre; più accessi per leggere tutto

Partizionamento e accorpamento

Il principio è che gli accessi si riducono **separando** attributi dello stesso concetto usati da operazioni diverse e **raggruppando** attributi di concetti diversi usati dalle stesse operazioni. Il **partizionamento verticale** divide gli attributi di un'entità in due entità legate uno a uno (dati anagrafici e dati lavorativi di un impiegato); quello **orizzontale** divide le occorrenze (analisti e venditori con gli stessi attributi). L'**accorpamento** è l'operazione inversa e si fa di solito sulle associazioni uno a uno, raramente su quelle uno a molti, mai su quelle molti a molti. Gli **attributi multivalore** si eliminano creando una nuova entità legata a quella originale (i numeri di telefono di un'agenzia diventano entità Telefono).

Scelta degli identificatori principali

Per ogni entità con più identificatori se ne sceglie uno come chiave primaria. I criteri: escludere gli identificatori con attributi opzionali, preferire quelli usati più spesso dalle operazioni, preferire chiavi semplici a composte e interne a esterne. Se nessuno soddisfa i criteri si introduce un codice artificiale.

Traduzione verso il modello relazionale

La **traduzione standard** è sempre possibile: ogni entità diventa una relazione con gli stessi attributi e la stessa chiave; ogni associazione diventa una relazione con i propri attributi più le chiavi delle entità collegate, che formano la sua chiave e sono chiavi esterne. Per le associazioni molti a molti è l'unica possibilità. Per le altre cardinalità si possono fondere entità e associazione:

CASO	TRADUZIONE CONSIGLIATA
Molti a molti: E1 (0,n) R (0,n) E2	E1(K1, A1) E2(K2, A2) R(K1, K2, AR), con K1 e K2 chiavi esterne
Uno a molti: E1 (1,1) R (0,n) E2	E1(K1, A1, K2, AR) con K2 chiave esterna NOT NULL; E2(K2, A2)
Uno a molti con E1 (0,1)	come sopra ma K2 e AR ammettono NULL; in alternativa la relazione separata R(K1, K2, AR)
Identificazione esterna di E1 da E2	E1(K2, K1, A1): la chiave di E2 entra nella chiave di E1
Uno a uno con (1,1) e (1,1)	si fonde l'associazione con una delle due entità (o le si fonde tutte e due); la chiave dell'altra diventa chiave esterna UNIQUE
Uno a uno con (0,1) e (1,1)	si fonde l'associazione con l'entità che partecipa obbligatoriamente, per evitare nulli
Uno a uno con (0,1) e (0,1)	tre relazioni: R(K1, K2, AR) con K2 UNIQUE, per non introdurre nulli

Con identificazioni esterne in cascata si propagano le chiavi partendo dall'entità che non ha identificazioni esterne. Lo schema logico finale si presenta elencando le relazioni con le chiavi primarie sottolineate e le chiavi esterne indicate (con la relazione riferita), insieme ai vincoli non esprimibili: CHECK, regole aziendali e vincoli da realizzare con i trigger.

CAPITOLO 10

Normalizzazione

Le dipendenze funzionali, gli assiomi di Armstrong, la chiusura e le chiavi, le decomposizioni senza perdita e con conservazione delle dipendenze, le forme normali di Boyce Codd e terza.

Ridondanze e anomalie

Una relazione che mescola informazioni su concetti diversi presenta ridondanze. In StudentiEsami(Matricola, Nome, Telefono, Corso, Voto) il telefono di uno studente è ripetuto per ogni esame. Ne derivano tre **anomalie**:

- **di aggiornamento**: se il telefono cambia va modificato in tutte le righe dello studente;
- **di cancellazione**: cancellando gli esami di uno studente si perdono anche i suoi dati anagrafici;
- **di inserimento**: uno studente che non ha ancora sostenuto esami non si può inserire.

La soluzione è **decomporre** la relazione in relazioni più piccole, ciascuna dedicata a un concetto. La teoria della normalizzazione dice quando una relazione va decomposta e come farlo senza perdere informazioni.

Dipendenze funzionali

DEFINIZIONE

Sia R uno schema e X, Y insiemi di suoi attributi. X **determina funzionalmente** Y, $X \to Y$, se per ogni istanza valida r e per ogni coppia di ennuple t_1, t_2 di r, $t_1[X] = t_2[X]$ implica $t_1[Y] = t_2[Y]$.

Nell'esempio valgono Matricola → Nome, Telefono e Matricola, Corso → Voto. Come le chiavi, le dipendenze nascono dalla semantica e non da un'istanza. Notazione: A, B attributi; X, Y, Z insiemi; XY sta per $X \cup Y$ e ABC per {A, B, C}. La dipendenza $X \to Y$ è **banale** se $Y \subseteq X$. Il concetto di chiave si riformula così: K è superchiave di R se e solo se $K \to R$.

Dato un insieme F di dipendenze, F **implica logicamente** $X \to Y$ ($F \models X \to Y$) se ogni istanza che soddisfa F soddisfa anche $X \to Y$; per esempio $\{A \to B, B \to C\} \models A \to C$. La **chiusura** F^+ è l'insieme di tutte le dipendenze implicate da F.

Assiomi di Armstrong

Per calcolare le dipendenze implicate serve un calcolo sintattico. Gli assiomi di Armstrong sono tre regole, e $F \vdash X \to Y$ indica che $X \to Y$ si deduce da F applicandole un numero finito di volte:

REGOLA	ENUNCIATO
Riflessività	se $Y \subseteq X$ allora $X \to Y$
Aumento	se $X \to Y$ allora $XZ \to YZ$
Transitività	se $X \to Y$ e $Y \to Z$ allora $X \to Z$
Unione (derivata)	se $X \to Y$ e $X \to Z$ allora $X \to YZ$
Decomposizione (derivata)	se $X \to Y$ e $Z \subseteq Y$ allora $X \to Z$
Pseudotransitività (derivata)	se $X \to Y$ e $WY \to Z$ allora $WX \to Z$

Il calcolo è **corretto** (se $F \vdash X \to Y$ allora $F \models X \to Y$, perché ogni assioma preserva la soddisfazione) e **completo** (se $F \models X \to Y$ allora $F \vdash X \to Y$). La dimostrazione di completezza costruisce, se $X \to Y$ non è deducibile, un'istanza di due ennuple uguali su X^+ e diverse altrove, che soddisfa F ma non $X \to Y$. Grazie alla decomposizione e all'unione, ci si può sempre ridurre a dipendenze con un solo attributo a destra.

Chiusura di un insieme di attributi

Calcolare F^+ è costoso (con $F = \{A \to B_1, \ldots, A \to B_n\}$ contiene almeno 2^n dipendenze). Si calcola invece la chiusura di un insieme di attributi, $X^+ = \{A \mid F \vdash X \to A\}$, grazie al lemma fondamentale: **$F \vdash X \to Y$ se e solo se $Y \subseteq X^+$**.

```
X(0) := X
ripeti
    X(j+1) := X(j) ∪ { A | esiste Y → Z in F con A ∈ Z e Y ⊆ X(j) }
finché X(j+1) = X(j)        /* nessun attributo aggiunto */
X+ := X(j)
```

In pratica: si parte da X e, a ogni giro, si aggiunge la parte destra di ogni dipendenza la cui parte sinistra è già tutta contenuta nell'insieme, finché non cambia più nulla. Esempio con R(A, B, C, D, E) e $F = \{A \to B, B \to C, AD \to E\}$, calcolando $(AD)^+$:

PASSO	DIPENDENZA APPLICATA	PERCHÉ SI PUÒ APPLICARE	INSIEME OTTENUTO
0	nessuna	si parte da X	A D
1	$A \to B$	A è nell'insieme	A B D
2	$B \to C$	B è nell'insieme	A B C D
3	$AD \to E$	A e D sono nell'insieme	A B C D E

L'insieme finale contiene tutti gli attributi, quindi AD è superchiave. Per sapere se è anche una chiave si controllano i sottoinsiemi: $A^+ = ABC$ e $D^+ = D$ non contengono tutto, quindi AD è minimale, cioè una **chiave**.

Trovare tutte le chiavi

L'algoritmo parte da un'osservazione semplice: gli attributi che **non compaiono mai a destra** di una dipendenza non possono essere determinati da altri, quindi stanno in ogni chiave. Gli attributi che compaiono **solo a destra** non servono mai in una chiave.

1. calcola N = attributi che non compaiono in nessuna parte destra; ogni chiave contiene N;
2. calcola N^+: se contiene tutti gli attributi, N è l'unica chiave;

3. altrimenti prova ad aggiungere a N gli attributi che compaiono sia a sinistra sia a destra, uno alla volta, poi a coppie, e così via, scartando i candidati che contengono una chiave già trovata;
4. ogni candidato la cui chiusura contiene tutti gli attributi, e che non contiene chiavi già trovate, è una chiave.

ESEMPIO SVOLTO

R(A, B, C, D, E, F) con F = {A → B, BC → D, D → E, E → F}.

PASSO 1	**Attributi mai a destra.** Le parti destre sono B, D, E, F: non compaiono mai A e C, quindi ogni chiave contiene A e C. N = A C
PASSO 2	**Chiusura di N.** A → B aggiunge B; ora BC è nell'insieme, quindi BC → D aggiunge D; poi D → E aggiunge E e E → F aggiunge F. $(AC)^+$ = A B C D E F
PASSO 3	**Conclusione.** La chiusura contiene tutti gli attributi: N è superchiave, e poiché ogni chiave deve contenere A e C non può esisterne una più piccola. unica chiave: **AC**

Equivalenza e ricoprimento minimale

Due insiemi F e G sono **equivalenti** se $F^+ = G^+$. Per verificarlo si controlla che ogni dipendenza Y → Z di F sia implicata da G ($Z \subseteq Y^+$ calcolata con G) e viceversa.

RICOPRIMENTO MINIMALE

F è **minimale** se: (1) ogni parte destra è un solo attributo; (2) nessuna dipendenza è ridondante, cioè togliendone una si ottiene un insieme non equivalente; (3) nessun attributo a sinistra è ridondante, cioè sostituendo X → A con Z → A, $Z \subset X$, si ottiene un insieme non equivalente. Ogni insieme di dipendenze ha un ricoprimento minimale equivalente.

L'ordine dei passi conta: prima si spezzano le parti destre, poi si eliminano gli **attributi ridondanti a sinistra** (passo 3), e solo alla fine le **dipendenze ridondanti** (passo 2). Per sapere se A è ridondante in XA → B si controlla se $B \in X^+$; per sapere se X → A è ridondante si controlla se A sta nella chiusura di X calcolata senza quella dipendenza.

ESEMPIO SVOLTO

F = {A → BC, B → C, AB → D} su R(A, B, C, D).

PASSO 1	**Parti destre singole.** $A \to BC$ si spezza in due dipendenze. $A \to B, \quad A \to C, \quad B \to C, \quad AB \to D$
PASSO 2	**Attributi ridondanti a sinistra.** L'unica parte sinistra con più attributi è AB. Si prova a togliere B: serve sapere se A da solo determina D. $A^+ = A\ B\ C\ D$ (contiene D) quindi $AB \to D$ diventa $A \to D$
PASSO 3	**Dipendenze ridondanti.** Si prova a togliere $A \to C$: con le altre si ha ancora $A \to B$ e $B \to C$, quindi C si ottiene lo stesso per transitività. $A \to C$ è ridondante e si elimina
PASSO 4	**Risultato.** Nessun'altra dipendenza si può togliere. ricoprimento minimale: $\{A \to B, B \to C, A \to D\}$

Decomposizioni senza perdita

Una **decomposizione** di R è un insieme di schemi $\{R_1, \ldots, R_k\}$ la cui unione dà tutti gli attributi di R. Per ogni istanza r vale sempre $r \subseteq \pi_{R1}(r) \bowtie \ldots \bowtie \pi_{Rk}(r)$, ma non sempre l'uguaglianza: il join può creare ennuple **spurie**. Con R(A, B, C), $r = \{a_1b_1c_1, a_2b_1c_2\}$ e decomposizione in AB e BC, il join contiene anche $a_1b_1c_2$, che non stava in r.

DECOMPOSIZIONE SENZA PERDITA (LOSSLESS JOIN)

La decomposizione $\{R_1, \ldots, R_k\}$ di (R, F) **preserva i dati** se per ogni istanza r che soddisfa F $r = \pi_{R1}(r) \bowtie \pi_{R2}(r) \bowtie \ldots \bowtie \pi_{Rk}(r)$.

Caso di due relazioni: $\{R_1, R_2\}$ è senza perdita se e solo se F implica $R_1 \cap R_2 \to R_1 - R_2$ oppure $R_1 \cap R_2 \to R_2 - R_1$, cioè se gli attributi comuni sono superchiave di almeno una delle due.

Per più relazioni si usa l'**algoritmo della matrice** (chase). Si costruisce una tabella con una riga per ogni R_i e una colonna per ogni attributo A_j; nella cella si scrive a_j se A_j sta in R_i, altrimenti b_{ij}. Poi si applicano ripetutamente le dipendenze: per $X \to Y$, se due righe coincidono su X si rendono uguali su Y (se una delle due ha a_j anche l'altra diventa a_j; altrimenti si sceglie uno dei due simboli b). Se a un certo punto compare una riga di sole a, la decomposizione preserva i dati; se non si può più applicare nulla e la riga non c'è, non li preserva.

Esempio: R(A, B, C, D) con $F = \{A \to B\}$ e decomposizione in $R_1(A, B)$ e $R_2(A, C, D)$. La matrice iniziale ha una riga per componente:

	A	B	C	D
$R_1(A, B)$	a_1	a_2	b_{13}	b_{14}
$R_2(A, C, D)$	a_1	b_{22}	a_3	a_4

Le due righe coincidono su A (entrambe a_1), quindi $A \to B$ le obbliga a coincidere anche su B: la prima ha a_2, perciò b_{22} diventa a_2. La seconda riga diventa $a_1\ a_2\ a_3\ a_4$, tutta di a: la decomposizione è **senza perdita**.

	A	B	C	D
$R_1(A, B)$	a_1	a_2	b_{13}	b_{14}
$R_2(A, C, D)$	a_1	$\mathbf{a_2}$	a_3	a_4

Se invece F = {A → C}, la stessa regola porta b_{13} ad a_3 nella prima riga, che diventa a_1 a_2 a_3 b_{14}; nessun'altra dipendenza si applica e nessuna riga è tutta di a: la decomposizione **perde dati**.

Conservazione delle dipendenze

La **proiezione** di F su Z, $\pi_Z(F)$, è l'insieme delle dipendenze X → Y di F^+ con XY ⊆ Z. Si calcola prendendo ogni sottoinsieme Y di Z e aggiungendo Y → $(Y^+ \cap Z) - Y$. Una decomposizione **conserva le dipendenze** se l'unione delle proiezioni $G = \cup \pi_{Ri}(F)$ è equivalente a F: ogni vincolo si può allora controllare dentro una sola relazione, senza join. Per verificarlo, per ogni X → Y di F si calcola X^+ rispetto a G e si controlla che contenga Y.

Le due proprietà sono indipendenti. Con R(C, S, Z), città, strada e codice postale, e F = {CS → Z, Z → C}, la decomposizione SZ e CZ preserva i dati (l'intersezione Z determina CZ − SZ = C) ma non le dipendenze: $\pi_{SZ}(F)$ è vuota, $\pi_{CZ}(F)$ = {Z → C}, e da queste non si ricava CS → Z.

Forme normali

FORMA NORMALE DI BOYCE CODD (BCNF)

R è in BCNF se per ogni dipendenza non banale X → A di F^+, **X è una superchiave** di R. Se F ha parti destre singole basta controllare le dipendenze di F.

TERZA FORMA NORMALE (3NF)

R è in 3NF se per ogni dipendenza non banale X → A di F^+, **X è una superchiave** di R **oppure A è primo**, cioè appartiene a qualche chiave.

Ogni schema in BCNF è anche in 3NF, non viceversa. La BCNF elimina tutte le ridondanze dovute a dipendenze funzionali: se due ennuple coincidessero su X senza che X sia superchiave, il valore di A nella seconda si potrebbe prevedere dalla prima, cioè sarebbe ripetuto. La 3NF tollera qualche ridondanza: R(C, S, Z) con CS → Z e Z → C ha chiavi CS e SZ; Z → C viola la BCNF perché Z non è superchiave, ma C è primo, quindi lo schema è in 3NF.

CHE COSA SI PUÒ OTTENERE

- decomposizione senza perdita con componenti in BCNF: **sempre possibile**;
- decomposizione senza perdita, che conserva le dipendenze, con componenti in 3NF: **sempre possibile**;
- decomposizione che conserva le dipendenze con componenti in BCNF: **non sempre possibile** (CSZ ne è il controesempio).

Algoritmo di decomposizione in BCNF

Finché esiste una componente R_i che non è in BCNF a causa di una dipendenza X → A:

1. si calcola X^+ e si crea la componente $T' = X^+$, con le dipendenze proiettate;
2. si crea la componente $T'' = T_i - (X^+ - X)$, cioè R_i senza gli attributi determinati da X ma conservando X, che fa da collegamento;
3. si sostituisce R_i con le due nuove componenti e si ricomincia.

Ogni passo è senza perdita perché l'intersezione X determina tutta T'. Il risultato dipende dall'ordine con cui si scelgono le dipendenze, e può perdere qualche dipendenza. Ogni schema con due soli attributi è in BCNF.

ESEMPIO SVOLTO

R(A, B, C, D, E) con F = {A → B, B → C, AD → E}.

PASSO 1	**Chiave.** A e D non compaiono mai a destra; $(AD)^+$ = ABCDE. chiave: AD
PASSO 2	**Violazioni.** In A → B e in B → C la parte sinistra non è superchiave; AD → E invece va bene. violano la BCNF: A → B, B → C
PASSO 3	**Prima scomposizione, con B → C.** B^+ = BC diventa una componente; l'altra è R senza C (gli attributi determinati da B), ma con B, che fa da collegamento. R_1(B, C) con B → C R'(A, B, D, E) con A → B, AD → E
PASSO 4	**Seconda scomposizione, con A → B.** In R' la chiave è AD, quindi A → B viola ancora. A^+ ristretta a R' è AB. R_2(A, B) con A → B R_3(A, D, E) con AD → E
PASSO 5	**Risultato.** Ogni componente ha come parte sinistra la propria chiave: sono tutte in BCNF, e ogni dipendenza di F sta dentro una componente, quindi sono anche conservate. R_1(<u>B</u>, C) R_2(<u>A</u>, B) R_3(<u>A, D</u>, E)

Algoritmo di sintesi in 3NF

1. si calcola un **ricoprimento minimale** di F;
2. gli attributi che non compaiono in nessuna dipendenza formano una componente a parte;
3. se una dipendenza coinvolge tutti gli attributi, R è già in 3NF;
4. altrimenti si crea una componente XA per ogni dipendenza X → A, raggruppando quelle con la stessa parte sinistra (X → A_1, …, X → A_n danno la sola componente XA_1…A_n);
5. se nessuna componente contiene una chiave di R, se ne aggiunge una formata da una chiave: questo garantisce che la decomposizione sia anche senza perdita.

ESEMPIO SVOLTO

R(A, B, C, D, E) con F = {AB → C, C → A, C → D, D → E}.

PASSO 1	**Chiavi.** B non compare mai a destra, ma B^+ = B non basta. Aggiungendo un attributo alla volta: $(AB)^+$ e $(BC)^+$ contengono tutti gli attributi, $(BD)^+$ e $(BE)^+$ no. chiavi: AB e BC; attributi primi: A, B, C

PASSO 2	**Forma normale.** In $D \to E$ la parte sinistra non è superchiave e E non è primo. R non è in 3NF
PASSO 3	**Ricoprimento minimale.** Le parti destre sono già singole e non c'è niente di ridondante. $F = \{AB \to C, C \to A, C \to D, D \to E\}$
PASSO 4	**Una componente per ogni parte sinistra.** $C \to A$ e $C \to D$ hanno la stessa parte sinistra e formano una sola componente. $R_1(A, B, C)$ da $AB \to C$ $R_2(C, A, D)$ da $C \to A$ e $C \to D$ $R_3(D, E)$ da $D \to E$
PASSO 5	**Chiave.** R_1 contiene già la chiave AB, quindi non serve aggiungere una componente.
PASSO 6	**Verifica.** Tutte le componenti sono in 3NF. R_1 non è in BCNF: vi vale $C \to A$, con C che non è superchiave di R_1, ma A è primo e la 3NF lo ammette.

METODO: CHIAVI E FORME NORMALI

Per trovare le chiavi, stabilire se uno schema è in BCNF o 3NF ed eventualmente decomporlo, un metodo efficace, che produce decomposizioni in BCNF quando possibile senza perdere dipendenze:

- trova le chiavi partendo dagli attributi che non compaiono mai a destra;
- per ogni dipendenza controlla se la parte sinistra è superchiave (BCNF) o se la parte destra è prima (3NF): basta una violazione per rispondere "no", e va citata;
- decomponi creando una relazione per ogni gruppo di dipendenze con la stessa parte sinistra, più una relazione con la chiave; poi verifica che ogni componente sia in BCNF;
- scrivi ogni componente con le sue dipendenze proiettate, nella forma $R_1(C, D)$ $F_1 = \{C \to D\}$.

CAPITOLO 11

Organizzazione fisica e indici

Come il DBMS memorizza i dati su disco: blocchi, buffer, organizzazione dei file, indici primari, secondari e multilivello, alberi B, esecuzione delle interrogazioni e progettazione fisica.

Memoria principale e secondaria

I programmi possono lavorare solo su dati in memoria principale, ma le basi di dati devono stare in memoria secondaria per due motivi: le **dimensioni** e la **persistenza**. La memoria secondaria è organizzata in **blocchi** di lunghezza fissa (qualche KB), e le uniche operazioni possibili sono la lettura e la scrittura di un blocco intero; useremo blocco e pagina come sinonimi.

Un accesso a disco richiede il posizionamento della testina, la latenza di rotazione e il trasferimento: in media non meno di qualche millisecondo, cioè quattro o più ordini di grandezza più di un'operazione in memoria centrale. Nelle applicazioni con molti accessi e poche elaborazioni (**I/O bound**) il costo dipende quindi **solo dal numero di accessi a blocco**: è l'unità di misura che useremo in tutti gli esercizi.

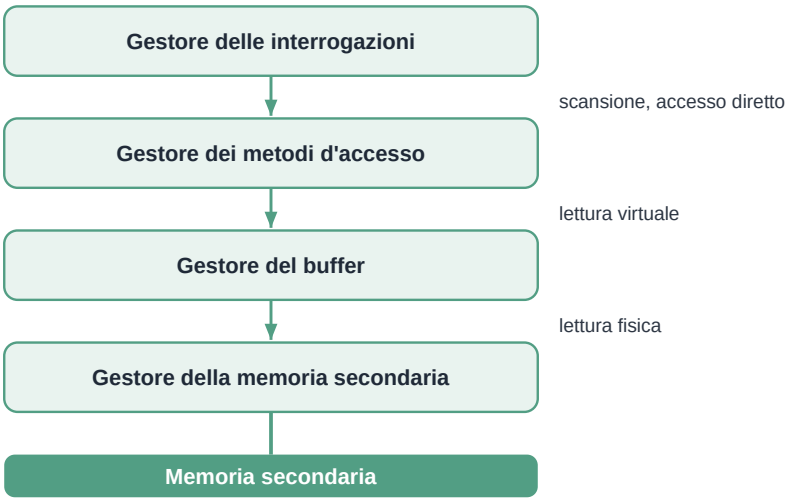


Figura 11.1. Il gestore degli accessi e delle interrogazioni: ogni livello offre servizi a quello superiore, fino alle letture e scritture fisiche dei blocchi.

Il gestore del buffer

Il **buffer** è un'area di memoria centrale gestita dal DBMS e condivisa dalle transazioni, organizzata in pagine grandi come i blocchi. Il suo scopo è ridurre gli accessi al disco: se una pagina richiesta è già nel buffer non serve leggerla, e le scritture si possono differire. Funziona bene grazie alla **località** dei dati e alla regola "80 e 20": l'80 per cento delle operazioni usa sempre lo stesso 20 per cento dei dati. Per ogni pagina il gestore tiene il file e il blocco di provenienza, un **contatore** dei programmi che la usano e un bit **dirty** che indica se è stata modificata. Le primitive sono:

- **fix**: richiede una pagina e incrementa il contatore; legge dal disco solo se la pagina non è nel buffer. Se non c'è una pagina libera, con la politica **steal** si sceglie una vittima, la si scrive se sporca e se ne usa lo spazio; con **no steal** la richiesta attende;
- **unfix**: la transazione ha finito di usare la pagina, il contatore scende;
- **setDirty**: segnala che la pagina è stata modificata;

- **force**: scrive la pagina su disco in modo sincrono, su richiesta del gestore dell'affidabilità.

Il DBMS usa il file system del sistema operativo solo per creare file e leggere o scrivere blocchi: l'organizzazione dei record nei blocchi e dei blocchi nei file la gestisce direttamente.

Record, blocchi e fattore di blocco

Il **fattore di blocco** bfr è il numero di record contenuti in un blocco. Con blocchi di B byte e record a lunghezza fissa di R byte, organizzati in modo **unspanned** (ogni record sta tutto in un blocco):

$$bfr = \lfloor B / R \rfloor \qquad b = \lceil r / bfr \rceil$$

dove r è il numero di record e b il numero di blocchi del file. Lo spazio residuo $B - bfr \cdot R$ di ogni blocco resta inutilizzato: con B = 120 e R = 25 si ha bfr = 4 e si sprecano 20 byte per blocco. In modo **spanned** un record può continuare nel blocco successivo, tramite un puntatore: si spreca meno spazio e si possono memorizzare record più grandi di un blocco, al prezzo di accessi più complicati.

Organizzazioni dei file

ORGANIZZAZIONE	CARATTERISTICHE	COSTO DI UNA RICERCA SU CHIAVE
Heap (seriale)	record nell'ordine di inserimento; inserimento in coda molto efficiente; molto diffusa insieme agli indici secondari	ricerca lineare: in media b / 2 accessi, b se la chiave non esiste
Ordinata (sequenziale)	record ordinati su un campo; inserimenti e cancellazioni costosi, gestiti con aree di overflow	ricerca binaria: $\lceil \log_2 b \rceil$ accessi
Hash	una funzione di hash sulla chiave calcola il blocco; collisioni gestite con blocchi di overflow collegati	poco più di 1 accesso per ricerche di uguaglianza; inutile per intervalli

Indici

Un **indice** è una struttura ausiliaria che velocizza l'accesso ai record in base ai valori di un campo, la **chiave dell'indice**, che non deve necessariamente essere identificante. L'idea è quella dell'indice analitico di un libro: un elenco ordinato di coppie (termine, pagina). Un indice è un file di record a due campi, **(valore della chiave, puntatore)**, ordinato sulla chiave.

- **indice primario**: costruito sul campo secondo cui il file è fisicamente ordinato; ce n'è al più uno;
- **indice secondario**: costruito su un campo diverso da quello di ordinamento; se ne possono avere molti;
- **indice denso**: ha una voce per ogni record (o per ogni valore della chiave);
- **indice sparso**: ha meno voci dei valori della chiave, tipicamente una per blocco.

Un indice primario può essere sparso: per ogni blocco del file contiene una voce k(i), p(i), dove p(i) è l'indirizzo del blocco e k(i) è la chiave del suo primo record, detto **ancora** del blocco. Il record con chiave k sta nel blocco p(i) tale che $k(i) \le k < k(i + 1)$. Un indice secondario invece deve essere **denso**, perché il file non è ordinato su quel campo e le ancore non servono.

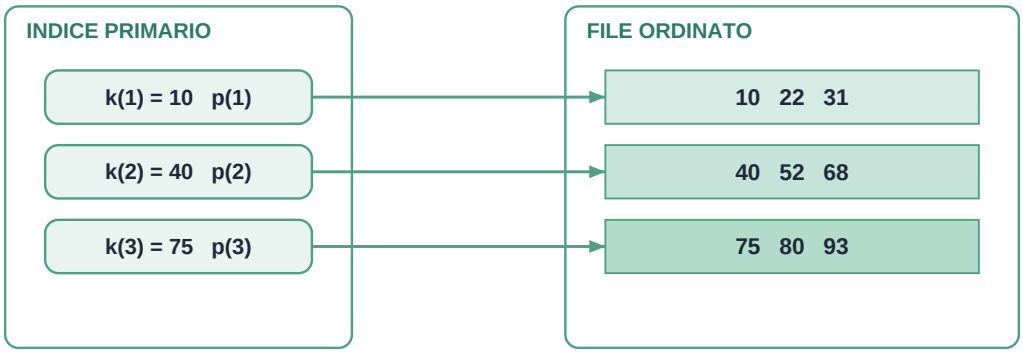


Figura 11.2. Indice primario sparso: una voce per blocco, con la chiave del primo record. Per cercare 52 si trova nell'indice la voce con 40 ≤ 52 < 75 e si legge il secondo blocco.

LE FORMULE PER GLI ESERCIZI

Siano V la dimensione del campo chiave e P quella del puntatore a blocco. Ogni voce dell'indice occupa V + P byte; il fattore di blocco dell'indice e il numero di blocchi dell'indice sono:

$$bfr_i = \lfloor B / (V + P) \rfloor \qquad b_i = \lceil r_i / bfr_i \rceil$$

dove r_i, il numero di voci, vale b (un blocco del file, una voce) per l'indice primario e r (un record, una voce) per l'indice denso o secondario. Una ricerca fa una ricerca binaria sull'indice e poi legge un blocco del file:

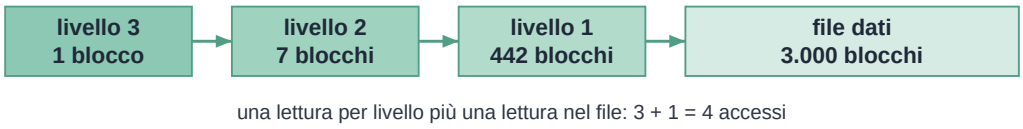
$$\text{accessi} = \lceil \log_2 b_i \rceil + 1$$

INDICE MULTILIVELLO

Un indice è a sua volta un file ordinato, quindi si può indicizzare. In un **indice multilivello** il primo livello è l'indice di partenza; il secondo è un indice primario sul primo, con una voce per ogni blocco del primo; e così via finché un livello sta in un solo blocco (il livello **top**). Tutte le voci hanno la stessa dimensione, quindi ogni livello riduce il numero di voci di un fattore fo = bfr_i, detto **fan out**:

$$b_1 = \lceil r_i / fo \rceil, \qquad b_2 = \lceil b_1 / fo \rceil, \qquad b_3 = \lceil b_2 / fo \rceil, \ \ldots \qquad \text{fino a } b_t = 1$$

La ricerca legge un blocco per livello e poi il blocco dei dati: **t + 1 accessi**. Il numero di livelli è circa $t = \lceil \log_{fo} r_i \rceil$: con fan out dell'ordine delle centinaia bastano tre o quattro livelli anche per centinaia di milioni di voci.



una lettura per livello più una lettura nel file: 3 + 1 = 4 accessi

Figura 11.3. Indice multilivello dell'esempio svolto: tre livelli sopra l'indice secondario denso.

ESEMPIO SVOLTO

Dati: r = 30.000 record, R = 100 byte, B = 1024 byte, organizzazione unspanned; campo chiave V = 9 byte, puntatore P = 6 byte. Si calcolano le grandezze una alla volta, sempre nello stesso ordine.

GRANDEZZA	FORMULA	CALCOLO	RISULTATO
record per blocco	$bfr = \lfloor B / R \rfloor$	$\lfloor 1024 / 100 \rfloor$	10
blocchi del file	$b = \lceil r / bfr \rceil$	$\lceil 30.000 / 10 \rceil$	3.000
voci per blocco di indice	$bfr_i = \lfloor B / (V + P) \rfloor$	$\lfloor 1024 / 15 \rfloor$	68

Con queste tre grandezze si confrontano le soluzioni:

SOLUZIONE	VOCI	BLOCCHI DELL'INDICE	ACCESSI
file disordinato, nessun indice			in media 3.000 / 2 = 1.500
file ordinato, nessun indice			$\lceil \log_2 3.000 \rceil = 12$
indice primario	b = 3.000	$\lceil 3.000 / 68 \rceil = 45$	$\lceil \log_2 45 \rceil + 1 = 6 + 1 =$ **7**
indice secondario denso	r = 30.000	$\lceil 30.000 / 68 \rceil = 442$	$\lceil \log_2 442 \rceil + 1 = 9 + 1 =$ **10**

Infine il multilivello sopra l'indice secondario: si divide per 68 finché resta un solo blocco.

PASSO 1	**Livello 1**: è l'indice denso di partenza. $b_1 = 442$ blocchi
PASSO 2	**Livello 2**: una voce per ogni blocco del livello 1. $b_2 = \lceil 442 / 68 \rceil = 7$ blocchi
PASSO 3	**Livello 3**: una voce per ogni blocco del livello 2; un solo blocco, è il livello top. $b_3 = \lceil 7 / 68 \rceil = 1$ blocco
PASSO 4	**Accessi**: un blocco per livello più il blocco dei dati. $t + 1 = 3 + 1 =$ **4** accessi

Alberi B e indici dinamici

Gli indici visti finora sono strutture ordinate statiche (come l'ISAM): ottimi per le ricerche, ma inserimenti e cancellazioni li degradano e richiedono aree di overflow e riorganizzazioni periodiche. I DBMS usano indici multilivello **dinamici** basati sugli **alberi B**: alberi di ricerca bilanciati in cui ogni nodo occupa una pagina e contiene molte chiavi ordinate con i puntatori ai figli. Ogni nodo tranne la radice è almeno pieno a metà, e gli algoritmi di inserimento e cancellazione dividono o fondono i nodi mantenendo l'albero **bilanciato**: tutte le foglie sono alla stessa profondità, quindi ogni ricerca costa un numero di accessi pari all'altezza, molto piccola grazie al fan out elevato. Nella variante **B+** i dati (o i puntatori ai record) stanno solo nelle foglie, collegate in sequenza, il che rende efficienti anche le ricerche per intervalli.

In MySQL con InnoDB la tabella è organizzata come albero B+ sulla chiave primaria (indice primario denso, creato automaticamente) e gli indici secondari sono alberi B+ densi. In SQL gli indici, pur non essendo nello standard, si definiscono ovunque in modo simile: CREATE [UNIQUE] INDEX Nome ON Tabella(Attributi) e DROP INDEX Nome.

Esecuzione e ottimizzazione delle interrogazioni

Il **query processor** riceve un'interrogazione SQL e la trasforma in un **piano di accesso**. I passi sono: analisi lessicale, sintattica e semantica con il catalogo; traduzione in algebra; ottimizzazione algebrica (capitolo 4); ottimizzazione basata sui costi. Quest'ultima usa i **profili** delle relazioni conservati nel catalogo: cardinalità, dimensione delle ennuple e dei valori, numero di valori distinti, minimo e massimo di ogni attributo, aggiornati periodicamente.

- **scansione**: lettura sequenziale di tutta la relazione, eventualmente con selezione e proiezione al volo;

- **accesso diretto con indice**: efficace per condizioni puntuali ($A = v$) e per intervalli ($v_1 \le A \le v_2$); con condizioni in AND si usa l'indice più selettivo e si verificano le altre in memoria; con condizioni in OR servono indici su tutte;
- **accesso diretto con hash**: efficace solo per uguaglianze;
- **join**: l'operazione più costosa, realizzata con **nested loop** (per ogni ennupla esterna si scorre l'interna, magari con un indice), **merge scan** (entrambe ordinate sull'attributo di join, si fondono) o **hash join** (entrambe partizionate con la stessa funzione hash).

L'ottimizzatore costruisce un albero di decisione con le alternative (quale accesso, quale ordine dei join, quale metodo), stima il costo di ciascun piano e sceglie il migliore; trova di solito una buona soluzione, non necessariamente l'ottima. Un indice su più campi, per esempio (cognome, nome), funziona per l'accesso su cognome ma non su nome solo; un hash su (cognome, nome) non funziona per nessuno dei due da solo.

Progettazione fisica

È l'ultima fase della progettazione: riceve lo schema logico e il carico applicativo e produce lo schema fisico, cioè le strutture di memorizzazione e i parametri specifici del DBMS. Nei sistemi relazionali coincide in buona parte con la **scelta degli indici**: le chiavi primarie sono indicizzate automaticamente; si aggiungono indici sugli attributi usati in selezioni e join frequenti e selettivi, tenendo conto che ogni indice rallenta gli aggiornamenti e occupa spazio. Se le prestazioni non bastano si "tara" il sistema aggiungendo o togliendo indici e verificando con EXPLAIN quali vengono usati davvero.

METODO: CONFRONTARE LE SOLUZIONI CON INDICI

Dati r, R, B, V e P, per confrontare il numero medio di accessi a blocco si procede sempre in quest'ordine:

PASSO 1	**Il file.** Record per blocco e numero di blocchi. $bfr = \lfloor B / R \rfloor$ $b = \lceil r / bfr \rceil$
PASSO 2	**L'indice.** Quante voci (V + P byte ciascuna) stanno in un blocco. $bfr_i = \lfloor B / (V + P) \rfloor$
PASSO 3	**Indice primario.** Una voce per blocco del file. $r_i = b, \quad b_i = \lceil b / bfr_i \rceil$ $accessi = \lceil \log_2 b_i \rceil + 1$
PASSO 4	**Indice denso.** Una voce per record. $r_i = r, \quad b_i = \lceil r / bfr_i \rceil$ $accessi = \lceil \log_2 b_i \rceil + 1$
PASSO 5	**Multilivello.** Si divide ripetutamente per bfr_i, arrotondando per eccesso, fino a 1 blocco. $accessi = numero\ di\ livelli + 1$

Attenzione agli arrotondamenti: il fattore di blocco si arrotonda per **difetto** (un record non si spezza), il numero di blocchi per **eccesso** (anche un blocco parziale va letto). Due esercizi completi sono nel capitolo 15.

CAPITOLO 12

Transazioni e affidabilità

Le proprietà acide delle transazioni e il gestore dell'affidabilità: log, checkpoint, dump, modalità di scrittura e procedure di ripresa dopo i guasti.

Transazioni

Una **transazione** è una unità di elaborazione con un inizio (start transaction) e una fine, all'interno della quale si esegue una e una sola volta **commit work**, per terminare correttamente, oppure **rollback work**, per annullare tutto. Un sistema che esegue transazioni per conto di molte applicazioni concorrenti si dice transazionale (OLTP). Un programma applicativo può contenere più transazioni in sequenza.

```
START TRANSACTION;
UPDATE ContoCorrente SET Saldo = Saldo + 10 WHERE NumConto = 12202;
UPDATE ContoCorrente SET Saldo = Saldo − 10 WHERE NumConto = 42177;
SELECT Saldo INTO A FROM ContoCorrente WHERE NumConto = 42177;
IF (A >= 0) THEN COMMIT WORK;       /* il bonifico va a buon fine */
            ELSE ROLLBACK WORK;     /* saldo negativo: si annulla tutto */
```

Le proprietà ACID

PROPRIETÀ	SIGNIFICATO	CHI LA GARANTISCE
Atomicità	la transazione si esegue tutta o per niente: un guasto prima del commit richiede di disfare (UNDO), un guasto dopo il commit può richiedere di rifare (REDO)	gestore dell'affidabilità
Consistenza	al termine i vincoli di integrità sono soddisfatti; durante l'esecuzione possono essere violati, ma se lo sono alla fine la transazione viene annullata	gestore dell'integrità
Isolamento	l'esecuzione concorrente produce lo stesso risultato di una esecuzione in sequenza; una transazione non espone i suoi stati intermedi, evitando l'effetto domino	gestore della concorrenza
Durabilità	gli effetti di una transazione che ha fatto commit non vanno persi, anche in caso di guasto	gestore dell'affidabilità

L'esito normale è il commit; l'abort può essere richiesto dall'applicazione ("suicidio") o imposto dal sistema per violazione di vincoli, problemi di concorrenza o guasti ("omicidio").

Il gestore dell'affidabilità e il log

Il gestore dell'affidabilità esegue i comandi transazionali e le procedure di ripristino dopo i guasti. Si appoggia alla **memoria stabile**, un'astrazione di memoria che non si guasta, realizzata con la ridondanza (dischi replicati, nastri). Lo strumento centrale è il **log**: un file sequenziale in memoria stabile, un "diario di bordo" che registra in ordine tutte le operazioni, come il filo di Arianna che permette di ripercorrere la strada.

RECORD DI LOG	SIGNIFICATO
B(T), C(T), A(T)	begin, commit e abort della transazione T
I(T, O, AS)	inserimento dell'oggetto O con stato finale AS (after state)
D(T, O, BS)	cancellazione di O, che aveva stato BS (before state)
U(T, O, BS, AS)	modifica di O dal valore BS al valore AS
CK(T1, …, Tn)	checkpoint, con l'elenco delle transazioni attive
DUMP	registra che è stata fatta una copia completa della base di dati

UNDO di un'azione su O: per update e delete si ricopia in O il before state, per insert si elimina O. **REDO**: per insert e update si ricopia l'after state, per delete si elimina O. Entrambe sono **idempotenti**: undo(undo(A)) = undo(A) e redo(redo(A)) = redo(A), proprietà essenziale se la ripresa stessa viene interrotta da un guasto.

LE REGOLE DI SCRITTURA DEL LOG

- **Write Ahead Log**: la parte before dei record di log si scrive prima della corrispondente modifica nella base di dati; così si può sempre disfare;
- **Commit Precedenza**: la parte after dei record di log si scrive prima del commit; così si può sempre rifare.

L'esito di una transazione è deciso irrevocabilmente nel momento in cui il record di commit viene scritto nel log in modo sincrono (force). Un guasto prima di quell'istante porta a disfare la transazione, un guasto dopo a rifarla se necessario. I record di abort si possono scrivere in modo asincrono.

QUANDO SI SCRIVE NELLA BASE DI DATI

MODALITÀ	COMPORTAMENTO	RIPRESA
Immediata	la base di dati può contenere valori di transazioni non ancora in commit	serve UNDO, non serve REDO
Differita	la base di dati riceve i valori solo dopo il commit	serve REDO, non serve UNDO
Mista	le scritture avvengono quando conviene al buffer manager	servono UNDO e REDO

Checkpoint e dump

Il **checkpoint** serve a non dover ripercorrere il log dall'inizio dei tempi, come la chiusura dei conti di fine anno di un'amministrazione. Nella versione più semplice: si sospende l'accettazione delle operazioni; si scrivono su disco (force) tutte le pagine sporche delle transazioni in commit; si scrive nel log, in modo sincrono, il record CK con gli identificatori delle transazioni attive; si riprende. Dopo un checkpoint è certo che i dati delle transazioni concluse sono su disco e che quelle "a metà strada" sono elencate nel CK. Il **dump** è una copia completa della base di dati (backup), fatta di solito a sistema fermo e conservata in memoria stabile.

Guasti e ripresa

I **guasti soft** (errori di programma, crash, cadute di tensione) fanno perdere la memoria centrale ma non quella secondaria: si fa una **ripresa a caldo**. I **guasti hard** danneggiano la memoria secondaria ma non quella stabile, quindi non il log: si fa una **ripresa a freddo**. Il modello è *fail stop*: al guasto il sistema si ferma, poi riparte con la procedura di ripresa.

RIPRESA A CALDO

1. si percorre il log a ritroso fino all'ultimo checkpoint;
2. si costruiscono gli insiemi: UNDO parte con le transazioni attive al checkpoint e REDO vuoto; percorrendo il log in avanti dal checkpoint, ogni B(T) aggiunge T a UNDO e ogni C(T) sposta T da UNDO a REDO;
3. si ripercorre il log all'indietro fino alla più vecchia azione delle transazioni in UNDO e REDO, **disfacendo** le azioni delle transazioni in UNDO;
4. si ripercorre il log in avanti **rifacendo** le azioni delle transazioni in REDO.

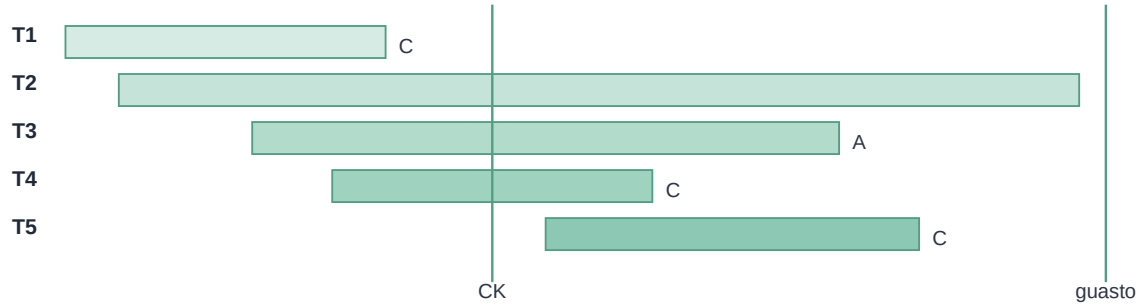


Figura 12.1. Le transazioni dell'esempio rispetto al checkpoint e al guasto. Al CK sono attive T2, T3 e T4. T4 e T5 fanno commit dopo il CK (REDO); T2 non termina e T3 fa abort (UNDO); T1 è conclusa prima del CK e non richiede nulla.

ESEMPIO SVOLTO

Il log contiene questi record; subito dopo l'ultimo avviene il guasto.

RECORD	SIGNIFICATO
U(T2, O1, B1, A1)	prima del checkpoint: T2 aggiorna O1 da B1 ad A1
U(T3, O2, B3, A3)	prima del checkpoint: T3 aggiorna O2 da B3 ad A3
U(T4, O3, B4, A4)	prima del checkpoint: T4 aggiorna O3 da B4 ad A4
CK(T2, T3, T4)	checkpoint: in quel momento sono attive T2, T3 e T4
C(T4)	T4 fa commit
B(T5)	inizia T5
U(T3, O3, B5, A5)	T3 aggiorna O3: prima valeva B5, dopo A5
U(T5, O4, B6, A6)	T5 aggiorna O4: prima valeva B6, dopo A6
D(T3, O5, B7)	T3 cancella O5, che valeva B7
A(T3)	T3 fa abort
C(T5)	T5 fa commit
I(T2, O6, A8)	T2 inserisce O6 con valore A8

Passo 1: costruire gli insiemi. Si parte dalle transazioni attive al checkpoint e si scorre il log in avanti.

RECORD LETTO	EFFETTO	UNDO	REDO
CK(T2, T3, T4)	le attive vanno in UNDO	T2, T3, T4	vuoto
C(T4)	T4 passa in REDO	T2, T3	T4
B(T5)	T5 entra in UNDO	T2, T3, T5	T4
A(T3)	nessuno: un abort va comunque disfatto	T2, T3, T5	T4
C(T5)	T5 passa in REDO	T2, T3	T4, T5

Passo 2: fase UNDO, all'indietro. Si disfano le azioni di T2 e T3 risalendo il log, anche prima del checkpoint, rimettendo il valore precedente (o eliminando ciò che era stato inserito).

AZIONE TROVATA	OPERAZIONE DI UNDO
I(T2, O6, A8)	si elimina O6
D(T3, O5, B7)	si reinserisce O5 = B7
U(T3, O3, B5, A5)	O3 = B5
U(T3, O2, B3, A3)	O2 = B3
U(T2, O1, B1, A1)	O1 = B1

Passo 3: fase REDO, in avanti. Si rifanno le azioni di T4 e T5 nell'ordine del log, scrivendo il valore successivo.

AZIONE TROVATA	OPERAZIONE DI REDO
U(T4, O3, B4, A4)	O3 = A4
U(T5, O4, B6, A6)	O4 = A6

RIPRESA A FREDDO

Si ripristinano i dati dall'ultimo dump; si rieseguono in avanti tutte le operazioni registrate nel log dal dump fino al guasto; infine si esegue una ripresa a caldo.

CAPITOLO 13

Controllo della concorrenza

Le anomalie delle esecuzioni concorrenti, gli schedule serializzabili (VSR e CSR), il locking a due fasi, lo stallo e i livelli di isolamento.

Un DBMS esegue decine o centinaia di transazioni al secondo e non può eseguirle una alla volta. Il **gestore della concorrenza** (scheduler) riceve le richieste di lettura e scrittura e decide se eseguirle, ritardarle o rifiutarle, in modo da evitare le anomalie. Modelliamo le transazioni come sequenze di letture r(x) e scritture w(x) su oggetti astratti x, y, z.

Le anomalie

ANOMALIA	CHE COSA SUCCEDE	CONFLITTO
Perdita di aggiornamento	r1(x) r2(x) w1(x) w2(x): due incrementi di x = 2 producono 3 invece di 4	scrittura e scrittura
Lettura sporca	t2 legge un valore scritto da t1, che poi fa abort: t2 ha visto uno stato mai esistito	lettura e scrittura con abort
Letture inconsistenti	t1 legge x due volte e trova valori diversi perché nel frattempo t2 lo ha modificato	lettura e scrittura
Aggiornamento fantasma	t1 legge y, t2 sposta 100 da y a z, t1 legge z: la somma y + z vista da t1 viola il vincolo	lettura e scrittura
Inserimento fantasma	t1 calcola due volte una media e trova risultati diversi perché t2 ha inserito una riga	su un dato "nuovo"

Schedule e serializzabilità

Uno **schedule** è una sequenza di operazioni di transazioni concorrenti, per esempio r1(x) r2(z) w1(x) w2(z). Ipotesi semplificativa: si considera la **commit proiezione**, cioè si ignorano le transazioni che fanno abort. Uno schedule è **seriale** se le transazioni non si mescolano; è **serializzabile** se produce lo stesso risultato di uno schedule seriale delle stesse transazioni. Serve dunque una nozione di equivalenza fra schedule.

VIEW SERIALIZZABILITÀ (VSR)

DEFINIZIONI

$r_i(x)$ **legge da** $w_j(x)$ se $w_j(x)$ precede $r_i(x)$ e non c'è un'altra scrittura di x fra le due. Se nessuna scrittura precede la lettura, si legge il valore iniziale.

$w_i(x)$ è una **scrittura finale** se è l'ultima scrittura di x nello schedule.

Due schedule sono **view equivalenti** se hanno la stessa relazione "legge da" e le stesse scritture finali. Uno schedule è **view serializzabile** (VSR) se è view equivalente a qualche schedule seriale.

Decidere se uno schedule è VSR è un problema NP completo: troppo costoso per uno scheduler reale. Nella pratica degli esercizi si ragiona così: dalle relazioni "legge da" e dalle scritture finali si ricavano vincoli di precedenza fra le transazioni; se i vincoli sono contraddittori lo schedule non è VSR, altrimenti si esibisce lo schedule seriale equivalente. Uno schedule con la perdita di aggiornamento, r1(x) r2(x) w1(x) w2(x), non è VSR: r1(x) e r2(x) leggono entrambe il valore iniziale, quindi in un seriale equivalente ciascuna transazione dovrebbe

precedere l'altra.

CONFLICT SERIALIZZABILITÀ (CSR)

Due azioni sono **in conflitto** se appartengono a transazioni diverse, operano sullo stesso oggetto e almeno una è una scrittura (conflitti rw, wr, ww). Due schedule sono **conflict equivalenti** se contengono le stesse operazioni e ogni coppia di operazioni in conflitto compare nello stesso ordine. Uno schedule è **conflict serializzabile** (CSR) se è conflict equivalente a uno schedule seriale.

TEOREMA DEL GRAFO DEI CONFLITTI

Si costruisce il **grafo dei conflitti**: un nodo per transazione e un arco da t_i a t_j se esiste un conflitto fra un'azione di t_i e una successiva di t_j. Uno schedule è in CSR **se e solo se** il grafo è **aciclico**; un ordinamento topologico del grafo dà lo schedule seriale equivalente.

Ogni schedule CSR è anche VSR, ma non viceversa: r1(x) w2(x) w1(x) w3(x) è view equivalente al seriale r1(x) w1(x) w2(x) w3(x) (w2 e w1 non sono lette da nessuno e la scrittura finale è w3), ma il grafo ha il ciclo t1 → t2 → t1. La differenza nasce dalle **scritture cieche**, cioè non precedute da una lettura.

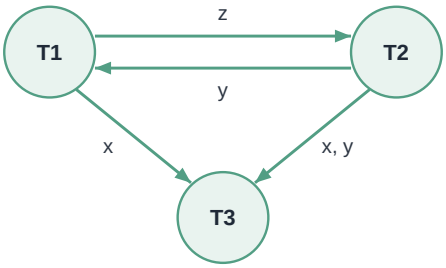


Figura 13.1. Grafo dei conflitti di r1(x) w2(y) r3(y) w1(z) r2(x) w3(x) r1(y) w2(z): il ciclo T1 → T2 → T1 dice che lo schedule non è CSR.

Locking a due fasi (2PL)

Costruire il grafo a ogni richiesta è impraticabile, e lo scheduler non conosce il futuro. In pratica si usano i **lock**: ogni lettura è preceduta da un **r_lock** (lock condiviso) e ogni scrittura da un **w_lock** (lock esclusivo), seguiti a tempo debito da unlock. Una transazione che prima legge e poi scrive lo stesso oggetto può chiedere subito il lock esclusivo oppure fare una **lock escalation** da condiviso a esclusivo. Il lock manager decide in base alla **tavola dei conflitti**:

RICHIESTA	RISORSA LIBERA	R_LOCKED	W_LOCKED
r_lock	concesso, r_locked	concesso, r_locked (contatore +1)	attesa
w_lock	concesso, w_locked	attesa	attesa
unlock	errore	contatore −1; libera a zero	libera

Se il lock non è concesso la transazione viene messa in attesa finché la risorsa non si libera. Il **locking a due fasi** aggiunge una regola sulle richieste: **una transazione, dopo aver rilasciato un lock, non può acquisirne altri**. Ogni transazione ha quindi una fase crescente, in cui acquisisce, e una calante, in cui rilascia.

Ogni schedule 2PL è CSR: ordinando le transazioni secondo l'istante in cui hanno tutti i lock e stanno per rilasciare il primo si ottiene uno schedule seriale equivalente. Il viceversa non vale: r1(x) w1(x) r2(x) w2(x) r3(y) w1(y) è CSR ma non 2PL, perché t1 dovrebbe rilasciare x prima di acquisire y. Il **2PL stretto** aggiunge che i lock si rilasciano solo dopo il commit o l'abort: elimina le letture sporche e rende superflua l'ipotesi della commit proiezione. È la versione usata dai DBMS e quella da simulare negli esercizi.

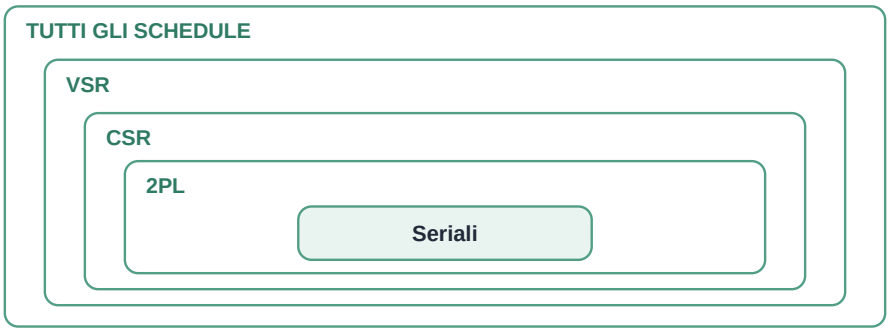


Figura 13.2. Le classi di schedule sono contenute l'una nell'altra: seriali ⊂ 2PL ⊂ CSR ⊂ VSR.

Lo stallo (deadlock)

Il locking introduce le **attese incrociate**: t1 legge x e poi vuole scrivere y, t2 legge y e poi vuole scrivere x. Con r_lock1(x), r_lock2(y), w_lock1(y), w_lock2(x) ognuna aspetta una risorsa tenuta dall'altra, all'infinito. Uno stallo corrisponde a un **ciclo nel grafo delle attese** (nodo = transazione, arco = attende). Le tecniche per gestirlo sono tre: il **timeout** (una transazione che aspetta troppo viene annullata; il problema è scegliere l'intervallo), il **rilevamento** (si cercano cicli nel grafo delle attese e si annulla una transazione) e la **prevenzione** (si annullano le transazioni "sospette" prima che lo stallo si formi, a costo di qualche annullamento inutile).

METODO: CSR, VSR E DEADLOCK CON IL 2PL

L'analisi di uno schedule si svolge in tre passi.

- **CSR**: per ogni oggetto elenca le operazioni nell'ordine; per ogni coppia in conflitto (transazioni diverse, almeno una scrittura) disegna l'arco dalla prima alla seconda. Se c'è un ciclo non è CSR, e basta citarlo.
- **VSR** (se non è CSR): scrivi le scritture finali e le relazioni "legge da" (incluse le letture del valore iniziale); ricava le precedenze obbligate e cerca una contraddizione, oppure costruisci il seriale equivalente.
- **2PL**: aggiungi il commit c_i dopo l'ultima operazione di ogni transazione; scorri lo schedule tenendo una tabella con r_lock e w_lock di ogni oggetto; una transazione bloccata mette in coda le sue operazioni successive; al commit rilascia tutto e le transazioni in attesa ripartono. Se tutte le transazioni non concluse sono in attesa l'una dell'altra c'è deadlock: indica il ciclo.

Altri due esercizi completi sono nel capitolo 15.

ESEMPIO: IL 2PL STRETTO PASSO PER PASSO

Schedule: r1(x) w2(y) r3(y) w1(z) r2(x) w3(x) r1(y) w2(z). Si legge un'operazione alla volta e si segna che cosa succede.

OPERAZIONE	ESITO	LOCK DOPO L'OPERAZIONE
r1(x)	concesso	x: lettura T1
w2(y)	concesso	x: lettura T1; y: scrittura T2
r3(y)	**T3 attende**: y è in scrittura a T2	invariati
w1(z)	concesso	x: lettura T1; y: scrittura T2; z: scrittura T1
r2(x)	concesso: la lettura è condivisa	x: lettura T1, T2; y: T2; z: T1
w3(x)	in coda: T3 è già ferma	invariati
r1(y)	**T1 attende**: y è in scrittura a T2	invariati
w2(z)	**T2 attende**: z è in scrittura a T1	invariati

Nessuna transazione può proseguire: T1 aspetta T2 e T2 aspetta T1. Il ciclo T1 → T2 → T1 nel grafo delle attese è un **deadlock** (T3, in attesa di T2, resta bloccata anche lei).

Livelli di isolamento

SQL:1999 permette di scegliere per ogni transazione il livello di isolamento, scambiando correttezza con prestazioni; una transazione può anche essere dichiarata di sola lettura. La perdita di aggiornamento è sempre evitata, perché sulle scritture si usa sempre il 2PL stretto.

LIVELLO	IMPLEMENTAZIONE	ANOMALIE ANCORA POSSIBILI
READ UNCOMMITTED	nessun lock in lettura, e non rispetta i lock altrui	letture sporche, letture inconsistenti, fantasmi
READ COMMITTED	lock in lettura rilasciati subito, senza 2PL	letture inconsistenti, aggiornamenti e inserimenti fantasma
REPEATABLE READ	2PL anche in lettura, lock sui dati	inserimento fantasma
SERIALIZABLE	2PL con lock di predicato (sull'indice o sull'intera tabella)	nessuna

Il **lock di predicato** blocca non solo le righe lette ma tutte quelle che soddisfano la condizione della query, comprese quelle che potrebbero essere inserite: è ciò che impedisce l'inserimento fantasma. Nel caso peggiore coincide con un lock sull'intera tabella; se esiste un indice sull'attributo della condizione basta bloccare l'intervallo dell'indice. In SQL il livello si sceglie prima della transazione:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
START TRANSACTION;
    SELECT AVG(Stipendio) FROM Impiegato WHERE Dip = 'A';   /* nessun fantasma */
COMMIT;
```

In MySQL con InnoDB il livello predefinito è REPEATABLE READ; i dettagli (SELECT … FOR UPDATE, FOR SHARE, autocommit) sono nella dispensa di laboratorio.

CAPITOLO 14

Basi di dati per XML

I dati semistrutturati: la sintassi di XML, i namespace, le DTD, la navigazione con XPath e le interrogazioni con XQuery.

XML e dati semistrutturati

XML (eXtensible Markup Language) è un metalinguaggio: non definisce un insieme fisso di marcatori, come HTML, ma le regole per crearne di propri. Deriva da SGML e organizza l'informazione in una struttura **gerarchica ad albero**, facile da navigare. I dati XML sono **semistrutturati**: la struttura è descritta dentro i dati stessi (self describing) e può variare da un elemento all'altro, a differenza delle tabelle relazionali in cui ogni riga ha lo stesso schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<Elenco>
  <Libro disponibilita="S">
    <Titolo>Il Signore degli Anelli</Titolo>
    <Autore>J.R.R. Tolkien</Autore>
    <Editore>Bompiani</Editore>
  </Libro>
  <Libro disponibilita="N">
    <Titolo>Il nome della rosa</Titolo>
    <Autore>Umberto Eco</Autore>
    <Editore>Bompiani</Editore>
  </Libro>
</Elenco>
```

Nel documento si riconoscono il **prologo**, cioè tutto ciò che precede l'elemento radice (la dichiarazione con version, obbligatoria, encoding e standalone, eventuali fogli di stile e il DOCTYPE), e l'**elemento radice**, unico, che contiene tutti gli altri.

REGOLE SINTATTICHE

- ogni elemento aperto va chiuso; un elemento vuoto si può scrivere <nome/>;
- gli elementi si annidano correttamente: si chiudono nell'ordine inverso a quello di apertura;
- i nomi di elementi e attributi distinguono maiuscole e minuscole;
- un documento che rispetta queste regole è **ben formato**; se rispetta anche una grammatica (DTD o XML Schema) è **valido**.

Gli **attributi** sono coppie nome e valore nel tag di apertura, per proprietà che non si vogliono mettere nel contenuto: il loro ordine non conta, uno stesso tag non può avere due attributi con lo stesso nome, il valore va sempre fra virgolette e non può contenere marcatori. I caratteri riservati si scrivono con le entità predefinite & < > ' ".

NAMESPACE

Se in un documento lo stesso nome ha significati diversi (il titolo di un libro e quello di un corso) si usano gli spazi di nomi: ogni namespace è identificato da un URI, che ha la forma di un indirizzo web ma serve solo a garantire l'unicità, ed è dichiarato con l'attributo xmlns. I nomi si scrivono prefisso:nome, come <libro:titolo> e <corso:titolo>; xmlns senza prefisso definisce il namespace predefinito.

DTD

Una DTD (Document Type Definition) descrive la grammatica di una classe di documenti: quali elementi esistono, che cosa contengono, quali attributi hanno. I simboli dopo un elemento indicano la cardinalità: + uno o più, * zero o più, ? zero o uno; #PCDATA indica testo; #REQUIRED un attributo obbligatorio.

```
<!DOCTYPE Artisti [
  <!ELEMENT Artista (nome, concerto+)>
  <!ELEMENT nome (#PCDATA)>
  <!ELEMENT concerto (data, luogo, numero_partecipanti)>
  <!ATTLIST Artista genere CDATA #REQUIRED>
]>
```

XPath

XPath individua parti di un documento con espressioni simili ai percorsi dei file, ed è la base di XSLT e XQuery. Un'espressione inizia di solito con doc("file.xml") e combina passi di navigazione:

SIMBOLO	SIGNIFICATO
/	radice, oppure figlio del nodo corrente
//	discendente a qualsiasi livello
. e ..	nodo corrente e nodo padre
@nome	attributo del nodo corrente
*****	qualsiasi elemento
[condizione]	predicato: tiene i nodi per cui la condizione è vera
[n]	posizione: l'n esimo nodo

```
doc("libri.xml")/Elenco/Libro                        (: tutti i libri :)
doc("libri.xml")//Autore                             (: autori a ogni livello :)
doc("libri.xml")/Elenco/Libro[Editore="Bompiani"]/Titolo
doc("libri.xml")/Elenco/Libro[2]/*                   (: figli del secondo libro :)
doc("libri.xml")//Libro[@disponibilita="S"]
```

XQuery e le espressioni FLWOR

XQuery è il linguaggio di interrogazione per XML definito dal W3C, "alla SQL", basato su XPath ed elaborazione di sequenze di nodi. Le interrogazioni sono **espressioni FLWOR**, dalle iniziali delle clausole:

CLAUSOLA	RUOLO	ANALOGO SQL
FOR	itera una variabile sui nodi di una sequenza: ogni nodo genera una tupla di legami	FROM
LET	lega una variabile a un'intera sequenza, senza moltiplicare le tuple	nessuno (una vista)
WHERE	filtra le tuple di legami	WHERE
ORDER BY	ordina le tuple	ORDER BY
RETURN	costruisce il risultato, una volta per tupla	SELECT

```
for $libro in doc("libri.xml")//Libro
where $libro/Editore = "Bompiani" and $libro/@disponibilita = "S"
order by $libro/Titolo
return <LibroBompiani>{ $libro/Titolo/text() }</LibroBompiani>
```

Nel RETURN si possono costruire nuovi elementi con espressioni annidate fra parentesi graffe; text() estrae il solo contenuto testuale. Le funzioni aggregate count, sum, avg, max e min operano sulle sequenze. Attenzione alla cardinalità: il RETURN viene eseguito una volta per ogni tupla generata dai FOR, mentre LET non ne aggiunge. Per iterare sui valori senza ripetizioni si usa distinct−values(), che elimina i duplicati. Sono disponibili anche le espressioni condizionali if, then, else.

```
(: il libro piu costoso, con titolo, autore e prezzo :)
let $max := max(doc("library.xml")/library/book/price)
for $b in doc("library.xml")/library/book
where $b/price = $max
return <result>
         <title>{ $b/title/text() }</title>
         <author>{ $b/author/text() }</author>
         <price>{ $b/price/text() }</price>
       </result>
```

METODO: INTERROGAZIONI XQUERY

Le richieste tipiche e lo schema da usare per ciascuna:

- **elementi con una condizione**: FOR sul percorso, WHERE con la condizione, RETURN;
- **il massimo o il minimo**: LET con max o min, poi un FOR con WHERE di uguaglianza, come nell'esempio;
- **totali e conteggi**: sum o count applicati a un percorso;
- **elenchi ordinati**: order by … descending;
- **date scritte come testo**: si convertono con xs:date prima di confrontarle.

XML e basi di dati

Esistono due approcci per memorizzare XML. Le **basi di dati XML native** usano un modello logico non relazionale e linguaggi specifici come XQuery, con strutture fisiche proprietarie, e sono adatte ai documenti orientati al testo (document centric). Le **basi di dati relazionali con supporto XML** memorizzano i documenti in tabelle, o come valori di colonne di tipo XML o scomponendoli secondo uno schema relazionale ricavato dalla DTD, e offrono estensioni come SQL/XML; sono adatte ai documenti orientati ai dati (data centric).

CAPITOLO 15

Esercizi svolti

Due esercitazioni complete, su schemi originali, che mettono insieme gli argomenti della dispensa: chiavi, algebra relazionale, SQL e trigger, progettazione, schedule, normalizzazione e indici.

Ogni esercitazione parte da uno schema relazionale e lo usa per una serie di domande, nell'ordine in cui gli argomenti compaiono nei capitoli. Per ciascuna domanda è indicato il metodo da applicare; conviene provare a risolverle da soli prima di leggere la soluzione.

DOMANDA	CONTENUTO	CAPITOLO
1a	chiavi primarie ed esterne di uno schema	2
1b	interrogazioni in algebra relazionale ("tutti", "il massimo")	3
1c	interrogazioni SQL e trigger	5, 7
1d, 1e	schema ER dello schema dato e analisi di una ridondanza	8, 9
2	schedule: CSR o VSR, deadlock con il 2PL	13
3	chiavi, forme normali e decomposizione	10
4	confronto fra indice primario, denso e multilivello	11

Esercitazione 1: festival musicali

```
Artista (id, nome, genere, nazione)
Festival (codice, nome, citta, data_inizio, data_fine, biglietti_venduti)
Esibizione (artista, festival, data, cachet)
Sponsor (id, nome, settore)
Sponsorizzazione (festival, sponsor, importo)
Biglietto (id, festival, tipo, prezzo, data_vendita)
```

1A. CHIAVI PRIMARIE ED ESTERNE

RELAZIONE	CHIAVE PRIMARIA	CHIAVI ESTERNE
Artista	id	nessuna
Festival	codice	nessuna
Esibizione	(artista, festival, data)	artista → Artista(id); festival → Festival(codice)
Sponsor	id	nessuna
Sponsorizzazione	(festival, sponsor)	festival → Festival(codice); sponsor → Sponsor(id)
Biglietto	id	festival → Festival(codice)

In Esibizione la data entra nella chiave perché lo stesso artista può esibirsi più sere nello stesso festival; se si esclude questa possibilità la chiave si riduce a (artista, festival).

1B.I ARTISTI CHE SI SONO ESIBITI IN TUTTI I FESTIVAL DI CATANIA

È lo schema "tutti", che si risolve con la divisione: a sinistra le coppie (artista, festival) realizzate, a destra l'elenco dei festival di Catania, con lo stesso nome di attributo.

PASSO 1	**Chi ha suonato dove.** Dalle esibizioni si tengono solo le coppie artista e festival. $R_1 = \pi_{\text{artista, festival}}(\text{Esibizione})$
PASSO 2	**I festival di Catania.** Si selezionano, si tiene il codice e lo si rinomina in festival, così che il nome coincida con quello di R_1. $R_2 = \delta_{\text{codice} \to \text{festival}}(\pi_{\text{codice}}(\sigma_{\text{citta = 'Catania'}}(\text{Festival})))$
PASSO 3	**La divisione.** Restano gli artisti che compaiono in R_1 insieme a *ogni* festival di R_2. $R_3 = R_1 \div R_2$
PASSO 4	**I nomi.** R_3 contiene solo gli id: con un join si recuperano i nomi. $\text{Risultato} = \pi_{\text{nome}}(\text{Artista} \bowtie_{\text{id = artista}} R_3)$

1B.II LO SPONSOR CON LA SPONSORIZZAZIONE DI IMPORTO MAGGIORE

Schema del massimo: si trovano le sponsorizzazioni superate da un'altra di importo più alto e si tolgono; quella che resta è la maggiore.

PASSO 1	**Due copie.** Per confrontare la tabella con sé stessa servono due copie con nomi diversi. $S_1 = \text{Sponsorizzazione}$ $S_2 = \text{Sponsorizzazione}$
PASSO 2	**Le sponsorizzazioni superate.** Il join tiene le coppie in cui S_1 ha un importo minore di S_2; si proiettano solo gli attributi di S_1. $S_3 = \pi_{\text{S1.festival, S1.sponsor, S1.importo}}(S_1 \bowtie_{\text{S1.importo < S2.importo}} S_2)$
PASSO 3	**La maggiore.** Togliendo le superate resta quella (o quelle, a pari merito) con l'importo massimo. $S_4 = S_1 - S_3$
PASSO 4	**Il nome dello sponsor.** $\text{Risultato} = \pi_{\text{nome}}(\text{Sponsor} \bowtie_{\text{id = sponsor}} S_4)$

1C.I PER OGNI FESTIVAL, IL NUMERO DI ARTISTI STRANIERI

```
SELECT f.codice, f.nome, COUNT(DISTINCT e.artista) AS artisti_stranieri
FROM Festival f
     JOIN Esibizione e ON e.festival = f.codice
     JOIN Artista a    ON a.id = e.artista
WHERE a.nazione <> 'Italia'
GROUP BY f.codice, f.nome;
```

COUNT(DISTINCT …) evita di contare due volte un artista che si esibisce più sere nello stesso festival.

1C.II TRIGGER: LE ESIBIZIONI DEVONO CADERE NEL PERIODO DEL FESTIVAL

```
CREATE TRIGGER ControllaDataEsibizione
BEFORE INSERT ON Esibizione
FOR EACH ROW
BEGIN
    DECLARE inizio DATE;
    DECLARE fine DATE;
    /* periodo del festival a cui si riferisce la nuova esibizione */
    SELECT data_inizio, data_fine INTO inizio, fine
    FROM Festival WHERE codice = NEW.festival;
    IF NEW.data < inizio OR NEW.data > fine THEN
        SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = 'Esibizione fuori dal periodo del festival';
    END IF;
END
```

Nella sintassi dello standard la stessa regola si scrive con WHEN (NEW.data NOT BETWEEN (SELECT data_inizio …) AND (SELECT data_fine …)). Va previsto anche un trigger gemello BEFORE UPDATE, perché la data può essere modificata.

1D. MANTENERE BIGLIETTI_VENDUTI IN FESTIVAL?

Volumi: 20 festival e 200.000 biglietti, quindi in media 200.000 / 20 = 10.000 biglietti per festival. Operazioni: O1, vendita di un biglietto, 2.000 volte al giorno; O2, lettura del numero di biglietti venduti per un festival, 50 volte al giorno. Si assume 1S = 2L.

PASSO 1	**Con la ridondanza, O1.** Si scrive il biglietto, poi si legge e si riscrive il festival con il contatore aumentato. Biglietto 1S + Festival 1L + Festival 1S = 2S + 1L = 5L 5L × 2.000 = 10.000L
PASSO 2	**Con la ridondanza, O2.** Il numero è già nel festival. Festival 1L × 50 = 50L
PASSO 3	**Senza la ridondanza, O1.** Si scrive solo il biglietto. Biglietto 1S = 2L 2L × 2.000 = 4.000L
PASSO 4	**Senza la ridondanza, O2.** Si leggono tutti i biglietti del festival per contarli. Biglietto 10.000L × 50 = 500.000L

CASO	O1 AL GIORNO	O2 AL GIORNO	TOTALE
Con ridondanza	10.000L	50L	**10.050L**
Senza ridondanza	4.000L	500.000L	**504.000L**

Conviene **mantenere** l'attributo: senza di esso ogni conteggio legge diecimila biglietti. Va tenuto aggiornato con un trigger AFTER INSERT (e AFTER DELETE) su Biglietto.

2. SCHEDULE

r1(x) w1(y) r2(y) r3(z) w2(z) r1(z) w3(x) r4(x) w4(y)

CSR, passo 1: le operazioni di ogni oggetto, nell'ordine. Un conflitto è una coppia di transazioni diverse in cui almeno una scrive; ogni conflitto dà un arco dalla prima alla seconda.

OGGETTO	OPERAZIONI IN ORDINE	ARCHI
x	r1 w3 r4	T1 → T3, T3 → T4
y	w1 r2 w4	T1 → T2, T1 → T4, T2 → T4
z	r3 w2 r1	T3 → T2, T2 → T1

CSR, passo 2: cercare un ciclo. Ci sono sia T1 → T2 (su y) sia T2 → T1 (su z): il ciclo T1 → T2 → T1 dice che lo schedule **non è CSR**.

VSR. Si guarda da chi legge ogni lettura:

PASSO 1	r2(y) legge il valore scritto da w1(y). quindi T1 deve venire prima di T2
PASSO 2	r1(z) legge il valore scritto da w2(z). quindi T2 deve venire prima di T1
PASSO 3	Le due condizioni sono incompatibili: nessuno schedule seriale le rispetta entrambe. **non è VSR**

2PL stretto. Si aggiunge il commit dopo l'ultima operazione di ogni transazione (c1 dopo r1(z), c2 dopo w2(z), c3 dopo w3(x), c4 dopo w4(y)) e si scorre lo schedule. Al commit una transazione rilascia tutti i suoi lock.

OPERAZIONE	ESITO	LOCK DOPO L'OPERAZIONE
r1(x)	concesso	x: lettura T1
w1(y)	concesso	x: lettura T1; y: scrittura T1
r2(y)	**T2 attende T1** (y in scrittura); w2(z) va in coda	invariati
r3(z)	concesso	x: lettura T1; y: scrittura T1; z: lettura T3
r1(z)	concesso: la lettura è condivisa	z: lettura T1, T3
c1	T1 rilascia x, y e z	z: lettura T3
r2(y)	T2 riparte: concesso	y: lettura T2; z: lettura T3
w2(z)	**T2 attende T3** (z in lettura a T3)	invariati
w3(x)	concesso: x è libero	x: scrittura T3; y: T2; z: T3
c3	T3 rilascia x e z	y: lettura T2
w2(z), c2	T2 riparte e termina	nessuno

OPERAZIONE	ESITO	LOCK DOPO L'OPERAZIONE
r4(x), w4(y), c4	senza attese	nessuno

Le attese sono solo T2 → T1 e poi T2 → T3, mai in cerchio: **non si verifica deadlock**. L'esecuzione reale è:

r1(x) w1(y) r3(z) r1(z) c1 r2(y) w3(x) c3 w2(z) c2 r4(x) w4(y) c4

3. NORMALIZZAZIONE

R(A, B, C, D, E, F, G, H) con F = {A → BC, C → D, EF → G, G → H}.

PASSO 1	**Attributi mai a destra.** Le parti destre sono B, C, D, G, H. ogni chiave contiene A, E, F
PASSO 2	**Chiusura di AEF.** A → BC aggiunge B e C; C → D aggiunge D; EF → G aggiunge G; G → H aggiunge H. $(AEF)^+$ = A B C D E F G H unica chiave: **AEF**
PASSO 3	**Forma normale.** In A → BC la parte sinistra A non è superchiave e B non è primo. non è in 3NF, quindi nemmeno in BCNF
PASSO 4	**Decomposizione.** Una componente per ogni dipendenza, più una che contiene la chiave.

COMPONENTE	NATA DA	CHIAVE	FORMA NORMALE
R_1(A, B, C)	A → BC	A	BCNF
R_2(C, D)	C → D	C	BCNF
R_3(E, F, G)	EF → G	EF	BCNF
R_4(G, H)	G → H	G	BCNF
R_5(A, E, F)	la chiave di R	AEF	BCNF

La decomposizione è senza perdita (R_5 contiene la chiave) e conserva tutte le dipendenze.

4. INDICI

r = 4.000.000 record, R = 250 byte, B = 4096 byte unspanned, V = 12 byte, P = 8 byte.

GRANDEZZA	CALCOLO	RISULTATO
record per blocco bfr	$\lfloor 4096 / 250 \rfloor$	16
blocchi del file b	$\lceil 4.000.000 / 16 \rceil$	250.000
voci per blocco di indice bfr_i	$\lfloor 4096 / (12 + 8) \rfloor$	204

SOLUZIONE	VOCI	BLOCCHI DELL'INDICE	ACCESSI
a) indice primario	250.000	$\lceil 250.000 / 204 \rceil = 1.226$	$\lceil \log_2 1.226 \rceil + 1 = 11 + 1 =$ **12**
b) indice denso	4.000.000	$\lceil 4.000.000 / 204 \rceil = 19.608$	$\lceil \log_2 19.608 \rceil + 1 = 15 + 1 =$ **16**

PASSO 1	**c) Multilivello dal primario, livello 1.** $b_1 = 1.226$
PASSO 2	**Livello 2.** $b_2 = \lceil 1.226 / 204 \rceil = 7$
PASSO 3	**Livello 3**, il top. $b_3 = \lceil 7 / 204 \rceil = 1$
PASSO 4	**Accessi.** 3 livelli + 1 = **4**

Per confronto, la ricerca binaria sul file ordinato costerebbe $\lceil \log_2 250.000 \rceil = 18$ accessi.

Esercitazione 2: clinica veterinaria

```
Proprietario (cf, nome, telefono, citta)
Animale (chip, nome, specie, data_nascita, proprietario, numero_visite)
Veterinario (matricola, nome, specializzazione, tariffa_oraria)
Ambulatorio (numero, piano, attrezzato_chirurgia)
Visita (id_visita, animale, veterinario, ambulatorio, data_ora, durata, costo)
Farmaco (codice, nome, prezzo)
Prescrizione (visita, farmaco, quantita)
```

1A. CHIAVI PRIMARIE ED ESTERNE

RELAZIONE	CHIAVE PRIMARIA	CHIAVI ESTERNE
Proprietario	cf	nessuna
Animale	chip	proprietario → Proprietario(cf)
Veterinario	matricola	nessuna
Ambulatorio	numero	nessuna
Visita	id_visita	animale → Animale(chip); veterinario → Veterinario(matricola); ambulatorio → Ambulatorio(numero)
Farmaco	codice	nessuna
Prescrizione	(visita, farmaco)	visita → Visita(id_visita); farmaco → Farmaco(codice)

1B. ALGEBRA: IL VETERINARIO CON LA TARIFFA PIÙ ALTA FRA QUELLI CHE HANNO VISITATO TUTTE LE SPECIE

Due schemi in sequenza: prima la divisione ("ogni specie"), poi il massimo sulla tariffa fra i veterinari rimasti.

PASSO 1	**Quali specie ha visitato ogni veterinario.** Il join con Animale porta la specie accanto a ogni visita. $V = \pi_{\text{veterinario, specie}}(\text{Visita} \bowtie_{\text{animale = chip}} \text{Animale})$
PASSO 2	**Tutte le specie.** $S = \pi_{\text{specie}}(\text{Animale})$
PASSO 3	**La divisione.** Restano i veterinari associati a ogni specie; si rinomina l'attributo per il join successivo. $T = \delta_{\text{veterinario} \to \text{matricola}}(V \div S)$
PASSO 4	**Le tariffe.** Si recuperano nome e tariffa dei veterinari trovati e se ne fa una seconda copia. $W_1 = \pi_{\text{matricola, nome, tariffa_oraria}}(\text{Veterinario} \bowtie T)$ $W_2 = W_1$
PASSO 5	**I superati.** Veterinari con una tariffa minore di qualcun altro del gruppo. $W_3 = \pi_{\text{W1.matricola, W1.nome, W1.tariffa_oraria}}(W_1 \bowtie_{\text{W1.tariffa_oraria < W2.tariffa_oraria}} W_2)$
PASSO 6	**Il massimo.** Si tolgono i superati. $\text{Risultato} = \pi_{\text{nome}}(W_1 - W_3)$

1C. VINCOLO: AL MASSIMO 8 VISITE AL GIORNO PER VETERINARIO

```
CREATE TRIGGER LimiteVisiteGiornaliere
BEFORE INSERT ON Visita
FOR EACH ROW
BEGIN
    DECLARE gia INT;
    /* visite gia fissate per lo stesso veterinario nello stesso giorno */
    SELECT COUNT(*) INTO gia FROM Visita
    WHERE veterinario = NEW.veterinario
      AND DATE(data_ora) = DATE(NEW.data_ora);
    IF gia >= 8 THEN
        SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = 'Il veterinario ha gia 8 visite in quel giorno';
    END IF;
END
```

In forma di asserzione: NOT EXISTS (SELECT veterinario, DATE(data_ora) FROM Visita GROUP BY veterinario, DATE(data_ora) HAVING COUNT(*) > 8).

1D. SCHEMA ER

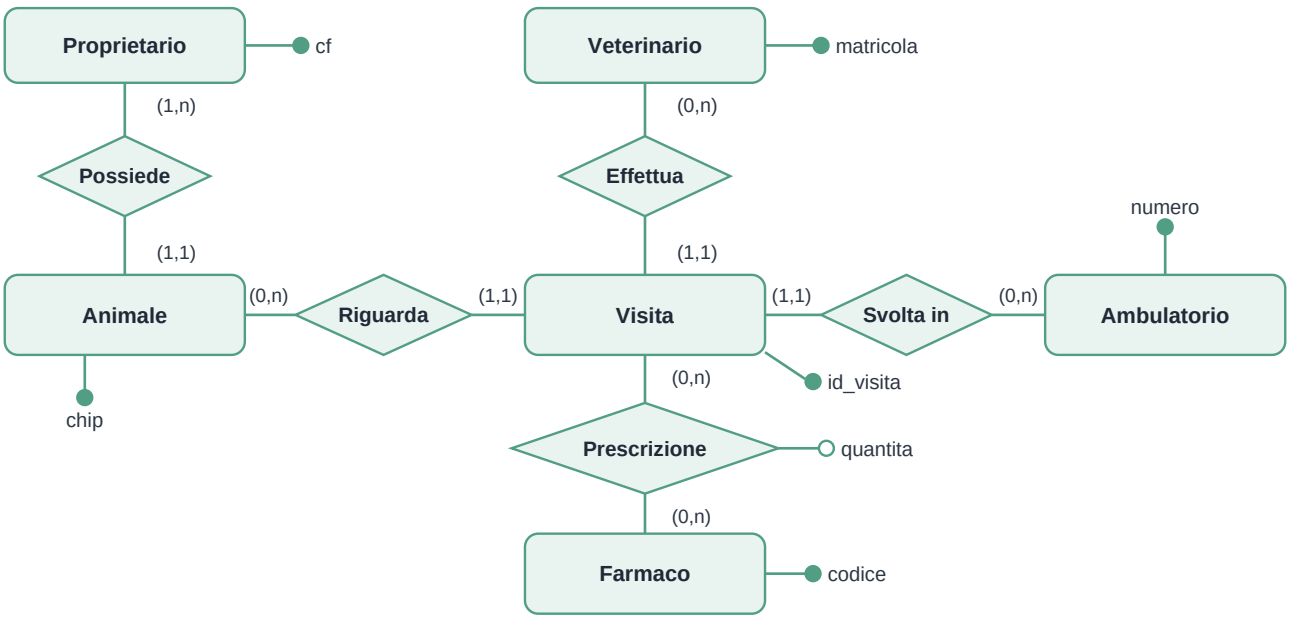


Figura 15.1. Schema ER ricostruito dallo schema relazionale della clinica. Sono indicati gli identificatori; gli altri attributi sono quelli dello schema. numero_visite è un attributo derivato di Animale.

Ogni chiave esterna diventa un'associazione uno a molti, con (1,1) dal lato della relazione che la contiene; Prescrizione, la cui chiave è formata da due chiavi esterne, diventa un'associazione molti a molti con l'attributo quantita. Il (1,n) di Proprietario esprime che si registrano solo proprietari con almeno un animale.

1E. MANTENERE NUMERO_VISITE IN ANIMALE?

Volumi: 5.000 animali e 60.000 visite, quindi in media 60.000 / 5.000 = 12 visite per animale. Operazioni: O1, nuova visita, 300 volte al giorno; O2, numero di visite di un animale, 30 volte al giorno. Si assume 1S = 2L.

PASSO 1	**Con la ridondanza, O1.** Si scrive la visita, poi si legge e si riscrive l'animale con il contatore aggiornato. Visita 1S + Animale 1L + Animale 1S = 5L 5L × 300 = 1.500L
PASSO 2	**Con la ridondanza, O2.** Animale 1L × 30 = 30L
PASSO 3	**Senza la ridondanza, O1.** Visita 1S = 2L 2L × 300 = 600L
PASSO 4	**Senza la ridondanza, O2.** Si contano le 12 visite dell'animale. Visita 12L × 30 = 360L

CASO	O1 AL GIORNO	O2 AL GIORNO	TOTALE
Con ridondanza	1.500L	30L	**1.530L**
Senza ridondanza	600L	360L	**960L**

Qui conviene **eliminare** l'attributo: le visite sono frequenti e il conteggio è raro e poco costoso. È il caso opposto a quello dell'esercitazione 1, dove ogni conteggio richiedeva diecimila letture: la decisione dipende sempre dal rapporto fra frequenze e volumi, non dalla natura dell'attributo.

2. SCHEDULE

r1(y) r2(x) w1(x) r3(z) w2(z) r3(y) w1(y) w3(x)

CSR, passo 1: le operazioni di ogni oggetto, nell'ordine.

OGGETTO	OPERAZIONI IN ORDINE	ARCHI
x	r2 w1 w3	T2 → T1, T2 → T3, T1 → T3
y	r1 r3 w1	T3 → T1 (r1 e w1 sono della stessa transazione)
z	r3 w2	T3 → T2

CSR, passo 2: cercare un ciclo. T1 → T3 (su x) e T3 → T1 (su y): il ciclo T1 → T3 → T1 dice che **non è CSR**.

VSR. Tutte le letture leggono il valore iniziale; le scritture finali sono x di T3, y di T1, z di T2.

PASSO 1	r3(y) legge il valore iniziale di y, che T1 scrive. quindi T3 deve venire prima di T1
PASSO 2	La scrittura finale di x è di T3, ma anche T1 scrive x. quindi T1 deve venire prima di T3
PASSO 3	Le due condizioni sono incompatibili. **non è VSR**

2PL stretto.

OPERAZIONE	ESITO	LOCK DOPO L'OPERAZIONE
r1(y)	concesso	y: lettura T1
r2(x)	concesso	x: lettura T2; y: lettura T1
w1(x)	**T1 attende T2** (x in lettura a T2)	invariati
r3(z)	concesso	x: lettura T2; y: lettura T1; z: lettura T3
w2(z)	**T2 attende T3** (z in lettura a T3)	invariati
r3(y)	concesso: la lettura è condivisa	y: lettura T1, T3
w1(y)	in coda: T1 è già ferma	invariati
w3(x)	**T3 attende T2** (x in lettura a T2)	invariati

Ora T2 attende T3 e T3 attende T2: è un **deadlock** fra T2 e T3, con T1 bloccata dietro T2.

3. NORMALIZZAZIONE

R(A, B, C, D, E, F, G) con F = {AB → C, C → D, D → B, E → F, F → G}.

PASSO 1	**Attributi mai a destra.** A ed E: stanno in ogni chiave. $(AE)^+$ = A E F G: non basta
PASSO 2	**Si aggiunge B.** AB → C aggiunge C, poi C → D aggiunge D. $(ABE)^+$ = tutti: chiave
PASSO 3	**Si aggiunge C.** C → D aggiunge D, D → B aggiunge B. $(ACE)^+$ = tutti: chiave
PASSO 4	**Si aggiunge D.** D → B aggiunge B, poi AB → C aggiunge C. $(ADE)^+$ = tutti: chiave
PASSO 5	**Si aggiunge F o G.** Si ottiene solo AEFG. nessuna chiave
PASSO 6	**Riepilogo.** chiavi: **ABE, ACE, ADE**; attributi primi: A, B, C, D, E
PASSO 7	**Forma normale.** In E → F la parte sinistra non è superchiave e F non è primo. non è in 3NF
PASSO 8	**Sintesi in 3NF.** F è già un ricoprimento minimale: una componente per dipendenza, più una con una chiave, perché nessuna componente ne contiene una.

COMPONENTE	NATA DA	CHIAVE	FORMA NORMALE
R_1(A, B, C)	AB → C	AB	3NF, non BCNF
R_2(C, D)	C → D	C	BCNF
R_3(D, B)	D → B	D	BCNF
R_4(E, F)	E → F	E	BCNF
R_5(F, G)	F → G	F	BCNF
R_6(A, B, E)	la chiave ABE	ABE	BCNF

La decomposizione è senza perdita e conserva le dipendenze. Tutte le componenti sono in BCNF tranne R_1: in essa vale C → B (da C → D e D → B), con C che non è superchiave di R_1; R_1 è comunque in 3NF perché B è primo. È un esempio del caso in cui BCNF e conservazione delle dipendenze non si possono avere insieme.

4. INDICI

r = 1.200.000 record, R = 300 byte, B = 2048 byte unspanned, V = 16 byte, P = 8 byte.

GRANDEZZA	CALCOLO	RISULTATO
record per blocco bfr	$\lfloor 2048 / 300 \rfloor$	6
blocchi del file b	$\lceil 1.200.000 / 6 \rceil$	200.000
voci per blocco di indice bfr_i	$\lfloor 2048 / (16 + 8) \rfloor$	85

SOLUZIONE	VOCI	BLOCCHI DELL'INDICE	ACCESSI
a) indice primario	200.000	$\lceil 200.000 / 85 \rceil = 2.353$	$\lceil \log_2 2.353 \rceil + 1 = 12 + 1 =$ **13**
b) indice denso	1.200.000	$\lceil 1.200.000 / 85 \rceil = 14.118$	$\lceil \log_2 14.118 \rceil + 1 = 14 + 1 =$ **15**

PASSO 1	**c) Multilivello dal primario, livello 1.** $b_1 = 2.353$
PASSO 2	**Livello 2.** $b_2 = \lceil 2.353 / 85 \rceil = 28$
PASSO 3	**Livello 3**, il top. $b_3 = \lceil 28 / 85 \rceil = 1$
PASSO 4	**Accessi.** 3 livelli + 1 = **4**

Lo spreco dovuto all'organizzazione unspanned è notevole: con 6 record da 300 byte restano inutilizzati 248 byte per blocco, più del 12 per cento dello spazio.