



Basi di Dati

Introduzione

• 3 sistemi informativi

Serve per rappresentare la parte del sistema di organizzazione di una qualsiasi azienda che mantiene le informazioni di interesse per quella organizzazione

Le informazioni devono essere organizzate per poter essere recuperate. → Il sistema informativo può essere o non essere automatizzato

La gestione automatizzata è più recente (da introduzione dei computer)

Il sistema informativo automatizzato ha come parte fondamentale la base di dati, che permette memorizzazione ed organizzazione dell'informazione

• de basi di dati

Insieme organizzato di dati utilizzato per rappresentare le informazioni di interesse → oggetti molto grandi che non possono essere presenti tutti nella memoria centrale, sopravvivono alle applicazioni che le adoperano e i dati presenti sono persistenti e di buona qualità

• Dati ed informazioni

Informazione: notizia, dato, elemento che consente di avere conoscenza di fatti

Nei sistemi automatizzati le informazioni diventano dati memorizzati in stringhe codificate. La decodifica dei dati permette di leggere le informazioni.

Codifica e decodifica avvengono secondo la stessa regola

La rappresentazione tramite simboli permette alla base di dati di avere una struttura stabile nel tempo che si mantiene anche con la modifica dei simboli.

→ Posso anche cambiare il modo in cui vado ad interrogare la base di dati e la risposta che ottengo è sempre consistente

• Condivisione delle informazioni

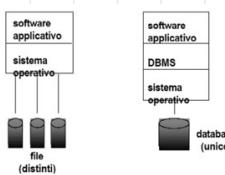
Essendo grande, la base di dati è condivisa. Ci sono dati strutturati, dati simbolici e dati condivisi. Essendo condivisa, ci sono varie applicazioni che lavorano sui dati. → Attenzione alle informazioni ripetute: creano incoerenza.

Base di dati risolve incoerenza, in quanto un'applicazione può usare una specifica parte, ma questo può anche creare problemi comuni a tutti gli applicativi

Vantaggi: evita le inconsistenze e permette di recuperare memoria

• DBMS

La gestione del database è affidata al DBMS → al di sopra al sistema operativo e permette la visione di un database su quella macchina rendendolo quindi relativo ad uno specifico sistema operativo. Le applicazioni accedono al database tramite le applicazioni.



→ Il DBMS è diverso dal filesystem del SO e contiene una visione unica dei dati messa a disposizione dei programmi (catalogo) ed utilizza le funzioni del filesystem per alcune funzionalità.

In questo modo hanno un livello di indipendenza alto, seppur a fronte di un costo elevato. Inoltre il DBMS è un software molto consistente essendo blocco unico gestito da una base centralizzata, ma allo stesso tempo permette anche di risparmiare memoria.

Caratteristiche fondamentali
→ **Privatezza:** i dati sono accessibili a chi ne ha diritto, essendo presenti meccanismi di autorizzazione differenti a seconda degli utenti.

→ **Efficienza** (nelle interrogazioni): DBMS non scorporabili perché alcune funzioni devono coesistere e devono usare le risorse per limitare il tempo di esecuzione derivante dall'utilizzo della memoria (non solo RAM).

→ **Efficacia:** permette di realizzare tutte le funzionalità in maniera utile e semplice per l'utilizzatore.

→ **Affidabilità:** garanzia che la base di dati rimanga intatta in ogni condizione rendendo possibile la ricostruzione in caso di guasto.

• de transazioni

→ Garanzia di affidabilità tramite transazioni: unità di lavoro base sulle basi di dati.

→ Viene cominciata da un utente e potrebbe anche portare i dati in condizioni di inconsistenza, a patto che il risultato sia consistente. Infatti ciò che è visibile all'esterno è solo lo stato iniziale prima della transazione e lo stato finale alla fine della transazione se terminata con successo. Essa può comprendere singole o sequenze di operazioni. → **Atomicità:** Molte operazioni vengono considerate come operazione unica

→ Essendo una sequenza, più operazioni possono lavorare sugli stessi dati → **concorrenza.** → La correttezza è garantita grazie alla serializzabilità, ovvero l'esecuzione in serie delle operazioni componenti la transazione.

Risultati vengono riportati sulla base di dati solo se conclusa con successo (commit)

Tipi di DBMS e organizzazione interna dei dati

DBMS: Access, Oracle, SQLServer

Record: insieme di campi che possono avere dati di tipo differente

Il gestore dei dati vede la rappresentazione degli stessi ed interagisce con gli utenti mostrando loro il modello logico dei dati, indipendente dalla realizzazione interna del database. → Sul modello logico l'utente costruisce la sua interrogazione (Esempio: sedie per rappresentare i posti di un cinema)

I modelli logici sono astratti e sono di vari tipi:

Gerarchici e reticolari: record con puntatori → molto vicini alla realizzazione interna

Relazionale: Unico costrutto è una relazione rappresentata da una tabella: ogni record è una riga di una tabella. Su queste tabelle è possibile scrivere le interrogazioni. → Ogni riga è un oggetto completo.

Lo schema: Struttura della relazione: che tipo di informazione ho nei vari campi

Istanze: insieme di valori che posso mettere all'interno di relazioni

Linguaggi utilizzati per i database

Può leggere una o più relazioni ed il risultato ottenuto è ancora una relazione.

I linguaggi utilizzati sono 2

Linguaggi di manipolazione (DML) che permettono di leggere e scrivere.

Linguaggi di definizioni (DDL) che permettono la definizione degli schemi. → Non sempre effettuate dallo stesso linguaggio.

Il linguaggio SQL comprende entrambe le parti ed è interattivo in quanto è possibile eseguire anche solo un'istruzione. È un linguaggio testuale, in quanto non ha interfaccia grafica e può essere usato in altri programmi.

L'architettura di un DBMS: two-tier

In generale si considera un'architettura distribuita con più macchine in rete (almeno 2): architettura client-server.

Server: accetta interrogazioni ed espone servizi: Passivi.

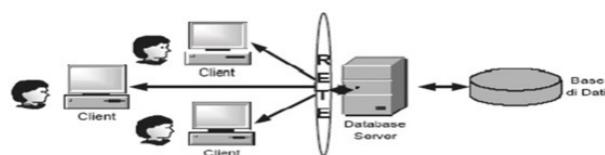
Client: macchine che fanno interrogazioni (richiedono i servizi): Attivi. → Client e server sono su macchine diverse

Linguaggio utente deve essere compreso dal server

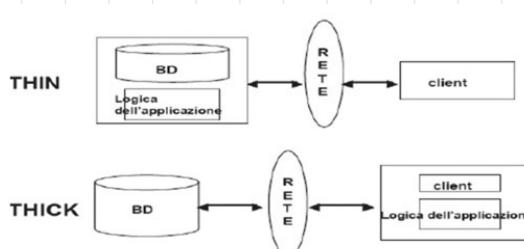
Struttura di base: Molti client e un unico server che gestisce la base di dati i cui servizi sono richiesti tramite la rete.

La macchina client può essere molto semplice (basta un browser), mentre la complessità della macchina server varia molto a seconda della varietà e vastità dei servizi da esso offerti.

L'architettura client-server viene detta two-tier.



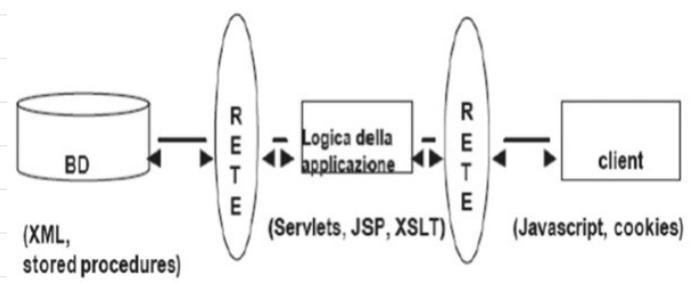
La struttura thin prevede che il client si occupi solo dell'interrogazione, mentre la struttura thick prevede che sul client siano presenti anche alcune parti del database, richiedendo un client più potente. In generale si usa la thin perché più affidabile.



L'architettura dei database: three tier

Essendo le macchine collegate tramite internet, si utilizza

un'architettura three tier. Quest'architettura permette di far coesistere sistemi eterogenei (con server di tipo differente ad esempio), i clienti sono thin ed è possibile aggiungere il numero di clienti a seconda della necessità aggiungendo macchine di livello intermedio. La macchina di livello intermedio svolge una parte di operazione (esempio: smista le richieste e le indirizza al database specifico)



3 big data

↳ Man mano che la base di dati viene utilizzata essa cresce, in quanto in generale i dati non scompaiono mai dalla base di dati.

↳ Si parla di Big Data, in quanto i dati hanno raggiunto dimensioni tali da introdurre il problema della loro gestione e memorizzazione (Esempio: Google con i dati delle posizioni degli utenti)

L'accesso inoltre è sempre più real-time (accesso immediato) e nuovi dati sono quelli dell'IoT, caratterizzato da sensori che forniscono una grande quantità di dati seppur di piccole dimensioni.

↳ Le caratteristiche dei big data sono 4:

→ Volume: I big data hanno la caratteristica di essere accompagnati da un'immensa quantità di interrogazioni nel corso di pochissimo tempo e le interrogazioni inoltre hanno un peso molto elevato nella trasmissione.



Per velocizzare le interrogazioni è necessario analizzare le interrogazioni. Si analizzano quindi dati filtrati precedentemente alla memorizzazione. Il filtraggio dei dati permette di memorizzare meno dati di quelli effettivamente trasmessi.

→ Velocità: I dati devono essere trasmessi in real-time.

→ Varietà: Il dato classico strutturato in tabelle non è quello più tipico in quanto vi sono molti contenuti multimediali non strutturabili in tabelle.

→ Veridicità: Nei big data è possibile inoltre ricavare informazioni vere anche se i dati non sono di buona qualità. → Data Quality.

La data science

↳ Statistiche sui big data.

↳ Permette di ottenere dati puliti (data cleaning) e combinare dati da varie sorgenti (data integration). Permette inoltre di ottenere informazioni utili dai dati (Data mining) grazie a varie tecniche. E' inoltre possibile utilizzare tecniche statistiche per fare previsioni. Queste tecniche sono associate alla probabilità.



All'interno della Data Science sono presenti anche le tecniche di apprendimento automatico (Machine Learning). Quest'ultimo si sta raffinando progressivamente, arrivando al Deep Learning, avvicinandosi sempre più al livello umano di pensiero.

La data engineering

↳ Gli ingegneri dei dati si concentrano sull'architettura dei dati e si preoccupano di integrare i dati e facilitare l'accesso al sistema.

Le tecnologie per i big data

↳ Alcune tecnologie sono state introdotte apposta per gestire i Big Data, come per esempio i sistemi di cloud computing, che permettono di elaborare molti dati in parallelo sfruttando il cloud. Un altro tipo di tecnologia pensata per i Big Data sono i sistemi NoSQL, dove non si gestiscono i dati con la complessità del linguaggio SQL, ma si pensa a dati con struttura molto semplice gestiti in maniera efficiente sacrificando in parte l'affidabilità dei sistemi transazionali.

→ Si passa da modello ACID a BASE.

↳ Garantiscono Disponibilità di base, che permettono di ottenere i dati in caso anche di fallimenti.

↳ La consistenza viene sacrificata e viene affidata all'utilizzatore. L'unica garanzia di consistenza è che in qualche momento i dati siano consistenti.

→ L'architettura di base è su 3 livelli e le applicazioni vengono eseguite nel cloud in modo da dimensionare la capacità di calcolo in base alle richieste.

→ Il database vero e proprio deve rispondere all'esigenza dei big data e dei molti utizzatori

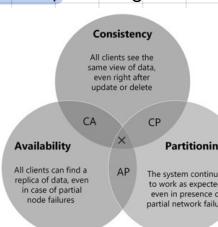
↳ Nel caso dell'approccio relazionale è necessario trovare dei server più potenti e non è possibile aggiungere server standard.

→ I sistemi che devono gestire dati non strutturati hanno sviluppato (proprietario) un database non relazionale.

→ Nell'approccio distribuito esiste il teorema CAP, che afferma che non possono essere mantenute contemporaneamente consistenza, partizionamento, e disponibilità ma se ne possono mantenere soltanto due.

→ Nei database relazionali si è scelto di mantenere consistenza e disponibilità dei dati.

→ Nei non relazionali è invece fondamentale la partition tolerance, mentre gli altri due aspetti sono a scelta del gestore dei database.



→ La scalabilità è orizzontale, in quanto basta aggiungere server, e per questo la scalabilità ha un costo non troppo alto.

→ Quando vengono aggiunti server i dati vengono distribuiti anche sui nuovi.

→ Per quanto riguarda l'affidabilità, questa è garantita dal fatto che i dati sono distribuiti sui vari server, e quindi sono recuperabili in ogni momento.

→ Si ha uno sharding, in quanto i dati vengono distribuiti in maniera trasparente tra i server senza che le applicazioni ne siano influenzate.

- C'è inoltre molta replicazione dei dati a differenza dei database relazionali.
 - ↳ La replicazione consente la sopravvivenza in caso di disastri, rendendo i database sempre disponibili.
- I database NoSQL permettono di avere Query efficienti nonostante la distribuzione.
- I sistemi NoSQL gestiscono il trasferimento da memoria secondaria a centrale come i database relazionali.
- I sistemi NoSQL sono semplici (più dei relazionali) e quindi scalabili in modo efficienti.
 - ↳ E' possibile scegliere il database più adatto alle applicazioni riducendo gli scambi con le applicazioni.
 - ↳ Sono assenti i controlli sull'integrità.
- Esempi sono MongoDB (eBay) e Cassandra (Facebook, Twitter).
 - ↳ MongoDB non mantiene la disponibilità, mentre Cassandra non mantiene la consistenza.
- Uno svantaggio ulteriore è l'assenza di uno standard per la struttura del database → Ogni database mette a disposizione le interfacce con cui le applicazioni Possono interagire con il database.
- I sistemi NoSQL non coprono quindi tutte le esigenze del mercato, ma sono utili solo in specifici ambiti.

Il modello relazionale

→ Fornisce agli utenti gli strumenti per fare le query.

Le relazioni:

→ Concetto matematico: insieme di elementi

↓
Esempio: schedina del totocalcio. In questo caso sono presenti due squadre e due valori numerici (2 stringhe e 2 interi) con notazione posizionale. La prima squadra per convenzione gioca in casa e risultati vengono assegnati nello stesso ordine.

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



→ Possono esserci risultati differenti anche se il dominio è lo stesso, nel caso precedente di tipo stringa.

→ Tra le n-uple (righe) della relazione non c'è nessun ordinamento.

→ Una relazione matematica è fatta prendendo un elemento da ciascun dominio numerato. → Sottoinsieme qualsiasi del prodotto cartesiano dei domini.

↳ In una relazione matematica ho un ordinamento: l'i-esimo valore proviene dall'i-esimo elemento.

→ La struttura del modello relazionale non è posizionale in quanto la relazione possiede uno schema che fornisce l'interpretazione del significato della colonna.

↳ Dare un significato alle n-uple rende inutile la notazione posizionale

Casa	Fuori	RetiCasa	RetiFuori
Juve	Lazio	3	1
Lazio	Milan	2	0
Juve	Roma	0	2
Roma	Milan	0	1

→ SCHEMA della RELAZIONE

→ Le relazioni vengono rappresentate con le tabelle. Una tabella rappresenta una relazione se

→ I valori di ogni colonna sono omogenei

→ Le righe sono tutte diverse tra loro

→ Le intestazioni delle colonne sono diverse

→ Rispetto ai modelli reticolari/gerarchici, nel modello relazionale i riferimenti tra gli elementi di relazioni diverse sono dati da valori uguali.

→ Il legame stabilito con valori uguali permette di mantenere l'indipendenza che garantisce la sopravvivenza a lungo termine della base di dati.

→ Essendo slegata dalla realizzazione, è possibile portare i dati da un sistema ad un altro.

→ Semplificata la gestione da parte dell'utente.

→ Lo schema di una relazione prevede il nome della relazione associato ad una serie di attributi.

→ Una tupla su un insieme di attributi X è una funzione che associa a ciascun attributo A in X un valore del dominio di A (riga).

→ T[A] rappresenta il valore di t su a.

→ Un'istanza di una relazione è un insieme di tuple → Un'istanza di una base di dati è l'insieme di tutte le istanze delle tuple.

→ Non è possibile fare relazioni di relazioni → Gli elementi delle tuple devono essere domini semplici.

Le informazioni mancanti

→ Essendo che le tuple hanno dimensione fissa, è possibile che alcuni valori della tupla non siano noti in qualche momento. → Dati non hanno significato.

→ Introdotto il valore nullo → Denota l'assenza del valore della tupla in quel momento.

→ Tutti i domini sono estesi con il valore NULL e quindi possono essere inseriti ovunque valori nulli.

→ Quando ci sono troppi valori nulli la tupla perde di significato.

→ 3 significati di null

- Valore che non esiste
- Valore non noto in un certo momento,
- Valore esistente ma non rilevante per l'applicazione della base di dati.

→ Non c'è modo di distinguere tra i significati.

esami	Studente	Voto	Corso
	NULL	30	NULL
	NULL	24	02
	9283	28	01

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

• I vincoli di integrità

→ E' necessario controllare che i valori inseriti siano corretti, ovvero deve rispettare delle proprietà definite dall'utente. Essi valgono per tutte le istanze.

→ Il controllo viene effettuato dal gestore della base di dati. → Predicato che associa ad ogni istanza il valore vero o falso



Il DBMS verifica che venga mantenuta l'integrità tramite i vincoli di integrità, che dipendono strettamente dalla realtà di interesse e valgono per tutte le istanze.

→ Il DBMS permette di fare relazioni solo se le proprietà sono rispettate. → Se tutte sono corrette il database è consistente.

• I vincoli interrelazionali → sulla SINGOLA RELAZIONE

→ 2 tipologie di vincoli: → Vincoli sui singoli domini (riguardano un singolo predicato ed i valori che esso possono assumere), che limitano il dominio

→ Vincoli su più domini (riguardano più domini contemporaneamente) sempre nella stessa tupla.

→ Riguardano i valori assunti dai predicati. → È possibile utilizzare espressioni booleane (AND, OR, NOT)

→ E' possibile fare piccole computazioni tra predicati.

→ Per i vincoli su più domini viene scelta invece una superchiave, ovvero un insieme di attributi scelto per identificare univocamente le tuple di una relazione.

→ Una chiave minimale è una superchiave il cui insieme di attributi è irriducibile.

→ Una superchiave esiste sempre, in quanto le tuple sono distinte ed al massimo è possibile selezionarle tutte, anche se non sempre è la scelta ottimale.

→ Le chiavi permettono di accedere alla base di dati e di correlare gli elementi tra relazioni diverse.

→ Una caratteristica di SQL è l'impossibilità di avere attributi nulli nella chiave primaria. → Necessario aggiungere un attributo nel caso sia assente

→ In una superchiave possono essere presenti valori nulli.

→ Gli attributi di chiave primaria vengono identificati da una sottolineatura.

→ I vincoli di chiave fanno parte delle dipendenze funzionali. → Insieme di proprietà che possono essere attribuite ad una relazione

↓
Dati due insiemi X ed Y, ogni volta che è presente un valore in X, è presente lo stesso valore in Y.

↓
Possono essere utilizzate per garantire alcune proprietà del database.

→ La verifica delle condizioni è un'operazione che ha un peso dipendente dalla grandezza del database.

• I vincoli interrelazionali → coinvolgono PIÙ RELAZIONI

→ Permettono di mantenere l'integrità di informazioni correlate ad altre informazioni.

→ Non sono simmetrici → Si va dalla tabella master a quella slave e non viceversa.

→ I valori nulli non compaiono nella tabella che stabilisce il vincolo, ma possono comparire nella tabella che deve sottostare ad un vincolo.

→ Nei vincoli interrelazionali se viene fatta una modifica nella tabella principale (master), questa modifica implica modifiche sulla tabella secondaria (slave).

• Database NoSQL

→ Non c'è dimensione fissa né in Cassandra né in MongoDB. → La dimensione della riga non è fissata a priori e si possono aumentare le colonne all'occorrenza.

→ Il vantaggio è che si accede una volta per avere tutte le informazioni presenti riguardo a quella chiave.

→ Lo svantaggio è la difficoltà di mantenere la consistenza.

L'algebra relazionale ed il calcolo relazionale

Definizione: algebra relazionale

Insieme di operatori, definiti su relazioni che hanno come risultato altre relazioni. È possibile anche combinare più operatori in modo da avere operazioni complesse



Un primo esempio di questi operatori sono i tipici operatori insiemistici, che però possono essere applicati solo quando le tuple hanno gli stessi attributi

- L'operatore di UNIONE: Date due relazioni r_1 ed r_2 definite sullo stesso insieme di attributi X , è indicata con $r_1 \cup r_2$ la relazione su X contenente le tuple che appartengono ad r_1 , r_2 , oppure ad entrambe
- L'operatore di INTERSEZIONE: Si indica con $r_1 \cap r_2$ ed è una relazione su X contenente le tuple che appartengono sia ad r_1 che ad r_2
- L'operatore di DIFFERENZA: Si indica con $r_1 - r_2$ ed è una relazione su X contenente le tuple che appartengono ad r_1 e NON ad r_2
- L'operatore di RIDEONOMINAZIONE: Sia r una relazione definita sull'insieme di attributi X e sia Y un insieme di attributi con la stessa cardinalità. Inoltre siamo A_1, A_2, \dots, A_m e B_1, B_2, \dots, B_n ordinamenti per gli attributi in X ed Y . Allora la rideonominazione $\rho_{A_1 \dots A_m / B_1 \dots B_n}(r)$ contiene una tupla t' per ciascuna tupla t in r , definita come: t' è una tupla su Y e $t'[B_i] = t[A_i] \forall i=1 \dots n$



E' possibile avere anche schemi diversi ma attributi con lo stesso dominio. In questo caso può essere interessante cambiare lo schema cambiando i nomi degli attributi.

↳ SQL permette di fare l'unione tra due relazioni senza applicare la rideonominazione, purché le relazioni abbiano attributi compatibili.

- L'operatore di SELEZIONE: Indicato da $\sigma_F(r)$ e ha come risultato le tuple che soddisfano F che è la condizione di SELEZIONE
 - Le condizioni di selezione: prevedono confronti tra attributi e attributi e costanti e possono essere semplici o complesse, ottenute combinando varie espressioni tramite i connettivi logici AND, OR, NOT, rispettivamente indicati da \wedge, \vee, \neg



Può quindi essere di tipo $A \Theta B$ oppure $A \Theta C$, dove

- Θ è l'operando di confronto
- A e B sono attributi di X che ha senso confrontare
- C è una costante compatibile con il dominio di A

→ Viene generata una nuova relazione con lo stesso schema che ha un numero di tuple diverso.

→ E' un risultato intermedio utile per calcolare il risultato finale e i dati non selezionati non vengono distrutti.

→ E' possibile che siano presenti valori nulli. La condizione analizzata è vera solo per valori non nulli in quanto i dati nulli potrebbero assumere un valore non compreso dalla condizione. Per analizzare valori nulli è necessario utilizzare IS NULL e IS NOT NULL.

- L'operatore di PROIEZIONE: Data una relazione $r(x)$ e un sottoinsieme Y di X , la proiezione di r su Y è l'insieme di tuple su Y ottenute dalle tuple di r considerando solo i valori su Y → $\pi_Y(r) = \{t[Y] | t \in r\}$
 - Potrebbe risultare in una relazione con meno tuple di quella di partenza se non viene selezionata la chiave in quanto le tuple uguali vengono ignorate perché devono essere tutte diverse essendo il risultato una relazione. → La proiezione ha al più tante tuple quante ne ha la relazione originale.
 - Sia rideonominazione che proiezione che rideonominazione lavorano sulla singola tabella modificandone la vista.
 - ↳ Le operazioni possono essere composte ma deve essere tenuto conto dell'ordine in cui esse sono fatte.

- Il PRODOTTO CARTESIANO: Il prodotto cartesiano tra R e Q produce una relazione che ha come schema l'unione degli schemi di R e Q e come tuple tutte le combinazioni possibili tra di esse

- Il JOIN NATURALE: Relazione definita su $X_1 X_2$ (unione insiemii $X_1 X_2$) come $r_1 \bowtie r_2 = \{t \in X_1 X_2 | t[X_1] \in r_1 \wedge t[X_2] \in r_2\}$
 - Il join naturale prevede che lo schema sia l'unione degli schemi.
 - Analizza gli attributi e crea tuple soltanto se ci sono attributi comuni.

↳ E' possibile che avvenga perdita di informazione. → Facendo il join, poi la proiezione e poi di nuovo il join si ottiene un risultato diverso da quello di partenza.

→ Join completo: ogni tupla degli operandi contribuisce al risultato

→ Join incompleto: alcune tuple non fanno parte del risultato. → Queste tuple sono chiamate dangling

→ È commutativo e associativo

→ Se gli insiemi di attributi sono uguali si ha l'intersezione

→ Se gli insiemi di attributi sono disgiunti si ha il prodotto cartesiano

- Il THETA JOIN: Prodotto cartesiano seguito da una selezione → Se F è un'uguaglianza si parla di equi-join

Per applicare il join il nome degli attributi deve essere lo stesso. Nel caso che non lo sia deve essere applicata la rideonominazione

Le **INTERROGAZIONI**: funzioni applicate a istanze di base di dati che producono relazioni.

Il gestore ha un **modulo specifico** deputato ad eseguire le interrogazioni, il **Query Processor**, che comprende 3 parti.

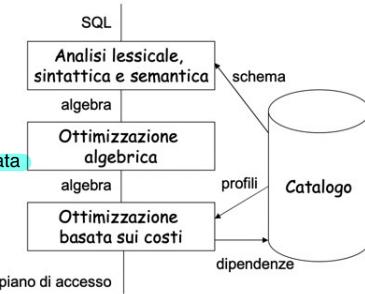
Nel primo modulo entra l'**istruzione** che viene esaminata e tradotta in un'espressione algebrica.



Sono presenti 2 ottimizzazioni: una **algebrica**, che prevede solo la traduzione della query in operatori algebrici ed una **basata sui costi**.

sui costi che utilizza i **profili delle relazioni** mantenuti nel **Catalogo** (dimensioni effettive dei dati).

Tutte le traduzioni sono per equivalenza, in quanto si mantiene il **funzionamento** durante le traduzioni.



L'equivalenza di espressioni

Producono lo stesso **risultato** ma vi arrivano attraverso **passaggi differenti**. L'equivalenza può dipendere dallo **schema** oppure essere **assoluta**.

$$\begin{aligned} \hookrightarrow \sigma_1 = \sigma_2 E_1 &\text{ se } \sigma_1(E_1) = \sigma_2(E_1) \text{ per ogni istanza } r \text{ di } R \longrightarrow \Pi_{AB}(R) \bowtie \Pi_{AC}(R) \equiv R \Pi_{ABC}(R) \text{ : Valida solo se l'intersezione tra gli attributi } a \text{ e } A \\ \hookrightarrow \sigma_1 = \sigma_2 &\text{ se } \sigma_1 \equiv \sigma_2 \text{ per ogni schema } R \longrightarrow \text{Esempio: } \Pi_{AB}(\sigma_{A>0}(R)) \equiv \sigma_{A>0}(\Pi_{AB}(R)) \end{aligned}$$

Utilizzata per **ottimizzare** le interrogazioni: vengono tradotte in algebra relazionale e ne viene **valutato** il costo utilizzando i **risultati intermedi**.

Viene scelta l'espressione che ha **costo minore** e **stesso risultato**: **trasformazioni di equivalenza**

→ **Atomizzazione delle selezioni**: Selezione con **espressione composta** può essere divisa in **selezioni semplici**

$$\hookrightarrow \sigma_{F_1 \wedge F_2}(E) = \sigma_{F_1}(\sigma_{F_2}(E))$$

→ **Idempotenza delle proiezioni**: una proiezione può essere **trasformata** in una **cascata di proiezioni** che **eliminano** gli **attributi** in **fasi successive**

$$\hookrightarrow \Pi_X(E) = \Pi_X(\Pi_{XY}(E))$$

→ **Anticipazione della selezione rispetto al join** (pushing selections down)

$$\hookrightarrow \sigma_F(E_1 \bowtie E_2) \equiv E_1 \bowtie \sigma_F(E_2) \text{ solo se la condizione } F \text{ si riferisce ad attributi di } E_2$$

→ **Anticipazione della proiezione rispetto al join** (pushing projections down)

$$\hookrightarrow \Pi_{X_1 Y_2}(E_1 \bowtie E_2) \equiv E_1 \bowtie \Pi_{Y_2}(E_2) \text{ solo se gli attributi di } X_1 - Y_2 \text{ NON sono coinvolti nel join}$$

→ Da idempotenza proiezioni: $\Pi_Y(E_1 \bowtie E_2) \equiv \Pi_Y(\Pi_{X_1}(E_1) \bowtie \Pi_{Y_2}(E_2)) \rightarrow$ **Eliminazione degli attributi NON coinvolti nel join e nel risultato finale**

→ **Inglobamento** di una **selezione** in un **prodotto cartesiano** a formare un **join**

$$\hookrightarrow \sigma'_F(E_1 \bowtie E_2) \equiv E_1 \bowtie E_2, \text{ dove l'intersezione tra gli insiemini degli attributi è vuota}$$

Esempio utilizzando le 5 proprietà:

$R_1(ABC), R_2(DEF), R_3(GHI)$, realizzazione di **SELECT A, E**

FROM R_1, R_2, R_3

WHERE $B > 100$ and $H = 7$ and $I > 2$ and $C = D$ and $F = G$

$$\Pi^{AE} (\sigma_{B>100} \text{ and } H=7 \text{ and } I>2 \text{ and } C=D \text{ and } F=G (R_1 \times R_2 \times R_3))$$

$$\downarrow \Pi^{AE} (\sigma_{B>100} (\sigma_{H=7} (\sigma_{I>2} (\sigma_{C=D} (\sigma_{F=G} (R_1 \times R_2 \times R_3))))))$$

$$\downarrow \Pi^{AE} (\sigma_{B>100} (R_1) \text{ JOIN } C=D \text{ } R_2 \text{ JOIN } F=G \text{ } \sigma_{I>2} (\sigma_{H=7} (R_3)))$$

$$\downarrow \Pi^{AE} (\underbrace{\Pi^{AF} (\sigma_{B>100} (R_1))}_{\text{risultato}} \text{ JOIN } C=D \text{ } R_2 \text{ JOIN } F=G \text{ } \Pi_G (\sigma_{I>2} (\sigma_{H=7} (R_3))))$$

risultato risultato necessario a join riduzione risultato intermedio

→ **Distributività della selezione rispetto all'unione**

$$\hookrightarrow \sigma'_F(E_1 \cup E_2) \equiv \sigma'_F(E_1) \cup \sigma'_F(E_2)$$

→ **Distributività della selezione rispetto alla differenza**

$$\hookrightarrow \sigma'_F(E_1 - E_2) = \sigma'_F(E_1) - \sigma'_F(E_2)$$

→ **Distributività della proiezione rispetto all'unione**

$$\hookrightarrow \Pi_X(E_1 \cup E_2) = \Pi_X(E_1) \cup \Pi_X(E_2)$$

$$\rightarrow \sigma'_F \cup \sigma'_G = \sigma'_F(R) \cup \sigma'_G(R)$$

$$\rightarrow \sigma'_F \cap \sigma'_G = \sigma'_F(R) \cap \sigma'_G(R) \equiv \sigma'_F(R) \bowtie \sigma'_G(R)$$

$$\rightarrow \sigma'_F \wedge \neg \sigma'_G(R) = \sigma'_F(R) - \sigma'_G(R)$$

$$\rightarrow E \bowtie (E_1 \cup E_2) = (E \bowtie E_1) \cup (E \bowtie E_2)$$

→ Proprietà commutativa e associativa operatori binari → **ATT perché era un test**

Il calcolo relazionale

→ Famiglia di linguaggi di interrogazione dichiarativa, ovvero che prevede solo la specifica delle proprietà del risultato e non della procedura utilizzata per giungervi



Semplificazioni e modifiche: i predici corrispondono alle relazioni, non compaiono simboli di funzione perché non necessari e compaiono prevalentemente formule aperte. → I valori dipendono dalle variabili libere e sono collegate alle interrogazioni in quanto definite tramite una formula di calcolo ed il risultato è costituito da tuple di valori che, sostituiti alle variabili libere, rendono vera la formula.

→ Calcolo relazionale su domini: forma $\{A_1: x_1, \dots, A_n: x_n | f\}$ → Prima della sbarra: target list

└─ A₁, ..., A_n sono attributi distinti che possono anche NON compatire nello schema della base di dati su cui viene effettuata l'interrogazione

x₁, ..., x_n sono variabili supposte distinte

f è una formula con le seguenti regole $\rightarrow R(A_1: x_1, \dots, A_p: x_p)$, dove R è uno schema di relazione e x₁, ..., x_p sono variabili

└─ $\exists y$ o $\forall c$ con x, y variabili e θ operatore di confronto

Ad f possono essere applicati anche gli operatori AND, OR, NOT ed i quantificatori esistenziale ed universale, lasciando inalterato il suo essere formula

└─ \exists : Proposizione vera solo se esiste un valore per il quale la condizione è vera

└─ \forall : Proposizione vera se per ogni valore del dominio la condizione è vera

Esempio:

IMPIEGATI (Matr, Nome, Eta, Stipendio) Trovare matricola, nome, età degli impiegati che guadagnano più di 60.000€

SUPERVISIONE (Capo, Impiegato)

ALGEBRA RELAZIONALE: $\exists x_1, x_2 (\text{Stipendio} > 60000 \text{ (IMPIEGATI)})$

CALCOLO RELAZIONALE: { Matr: m, Nome: n, Eta: e | $\exists s (\text{IMPIEGATI}(\text{Matr}: m, \text{Nome}: n, \text{Eta}: e, \text{Stipendio}: s) \wedge s > 60000)$ }

NON NECESSARIO in quanto SOTTINTESO → Utilizzati per DIFFERENZE

Trovare capi per i quali NON esiste un impiegato che guadagna più di 60.000€ → Sottrazione

{ Matr: c, Nome: m | IMPIEGATI (Matr: c, Nome: m, Eta: e, Stipendio: s) \wedge SUPERVISIONE (Impiegato: m, Capo: c) $\wedge \forall m' \forall e' \forall s' (\neg \text{IMPIEGATI}(\text{Matr}: m', \text{Nome}: m', \text{Eta}: e', \text{Stipendio}: s') \wedge$

SUPERVISIONE (Impiegato: m', Capo: c) $\vee s' > 60000))$ }

→ Era possibile usare $\neg \exists$ anziché \forall

→ Pregi e difetti del calcolo relazionale

→ $\{A_1: x_1, A_2: x_2 | R(A_1: x_1) \wedge x_1 = x_2\}$ → A₂ è un qualsiasi valore del dominio, e se questo è infinito anche i valori che possono essere assunti da A₂ sono infiniti, facendo di perdere di senso all'espressione



Un'espressione è indipendente dal dominio se il suo risultato non cambia variando il dominio sul quale è calcolata

└─ Il calcolo relazionale non è indipendente dal dominio, mentre l'algebra relazionale lo è



Algebra e calcolo relazionale sono equivalenti solo se vengono considerate espressioni indipendenti dal dominio

→ Il calcolo relazionale richiede molte variabili, il che lo rende particolarmente verboso

→ Un pregio è rappresentato dalla dichiaratività

Le chiusure transitività (interrogazione internamente ricorsiva)

→ In algebra si traduce con un numero di join illimitato e non esiste la possibilità di rappresentare la chiusura transitiva

└─ Esempio

Supervisione(Impiegato, Capo) → Per ogni impiegato, trovare tutti i superiori

Il calcolo su tuple con dichiarazione di range

↪ {T | Z | f}

└─ T è la target list, con elementi del tipo Y:x:Z, dove x è una variabile ed Y e Z sono sequenze di attributi. Gli attributi di Z devono comparire nella relazione che costituisce il range di x. Per selezionare tutti gli attributi è possibile usare x.*

└─ L è la range list dove vengono elencate le variabili libere della formula f con i relativi range. Le variabili sono del tipo x(R), dove R è una relazione

└─ f è una formula nella quale gli attributi possono essere confrontati con costanti oppure attributi di altre relazioni e possono essere presenti connettivi e quantificatori



Utilizzando le variabili, le dichiarazioni di range specificano che esse possono assumere valori solo nelle tuple della relazione associata e ciò le rende indipendenti dal dominio e non sono necessarie le condizioni del calcolo sui domini

Svantaggio: il calcolo su tuple non permette di rappresentare operazioni i cui risultati provengono da relazioni diverse in quanto i range delle variabili utilizzate per il risultato provengono da una singola relazione. Per questo motivo l'unione non è rappresentabile, mentre è possibile rappresentare intersezione e differenza nei modi

che seguono → INTERSEZIONE: $T_{BC}(R_1) \cap T_{BC}(R_2) = \{x_1.BC | x_1(R_1) \wedge \exists x_2(R_2)(x_1.B = x_2.B \wedge x_1.C = x_2.C)\}$

→ DIFFERENZA: $T_{BC}(R_1) - T_{BC}(R_2) = \{x_1.BC | x_1(R_1) \wedge \neg \exists x_2(R_2)(x_1.B = x_2.B \wedge x_1.C = x_2.C)\}$

Esempi:

① Matricola, nome, età e stipendio degli impiegati che guadagnano più di 40.000 €
↳ $\{i | i(\text{Impiegati}) \mid i.\text{Stipendio} > 40\}$

② Visualizzare solo matricola, nome ed età
↳ $\{i.\text{Matr}, \text{Nome}, \text{Eta} | i(\text{Impiegati}) \mid i.\text{Stipendio} > 40\}$

③ Trovare le matricole dei capi degli impiegati che guadagnano più di 40.000€
↳ $\{s.\text{Capo} | i(\text{Impiegati}), s(\text{Supervisore}) \mid i.\text{Matr} = s.\text{Impiegato} \wedge i.\text{Stipendio} > 40\}$

④ Trovare nome e stipendio dei capi degli impiegati che guadagnano più di 40.000€
↳ $\{\text{Nome}, \text{Stip} : i'(\text{Nome}, \text{Stip}) | i'(\text{Impiegati}), s(\text{Supervisore}), i(\text{Impiegati}) \mid i'.\text{Matr} = s.\text{Capo} \wedge s.\text{Impiegato} = i.\text{Matr} \wedge i.\text{Stipendio} > 40\}$

Algebra relazionale estesa

→ Il JOIN ESTERNO: Il join esterno permette di non perdere informazioni quando viene fatto il join tra tabelle. Questo può essere utile in quanto quando viene fatto un join naturale alcune parti possono andar perse. Il left outer join permette di mantenere tutte le righe della relazione di sinistra e mette a NULL le tuple che non fanno join sulla relazione di destra. Il right outer join fa il contrario. Viene utilizzato in alcuni casi per fare la differenza

Simboli: FULL = Δ , LEFT = \bowtie , RIGHT = \bowtie^*

→ La PROIEZIONE GENERALIZZATA $\rightarrow \Pi_{F_1, F_2, F_3}(E)$: F_1, F_2, F_3 sono espressioni aritmetiche su attributi di E e costanti. Permette quindi operazioni direttamente sugli attributi che verranno proiettati

→ Funzioni AGGREGATE: Prendono in ingresso una relazione e producono un valore numerico dato da operazioni riguardanti più tuple

↳ Sum Attributo (Relazione) → Restituisce la somma dei valori assunti da un determinato attributo

↳ Count Attributo (Relazione) → Conta le occorrenze di un attributo

↳ Max Attributo (Relazione) → Restituisce il valore massimo di un attributo

↳ Count-distinct Attributo (Relazione) → Conta soltanto le occorrenze diverse

→ Il raggruppamento → Operatore G

↳ Esempio: cliente G sum(credito) (Conto)
↳ Clienti: attributo su cui viene fatto il raggruppamento
↳ Sum = funzione aggregata applicata a credito
↳ Conto = relazione a cui si applica tutto

→ L'operatore DIVISIONE

→ È utile per interrogazione di tipo "universale" (quelle che richiedono "tutto"). È un operatore derivato in quanto può essere ricondotto ad operatori fondamentali quali la sottrazione

→ Esempio: Trovare i nomi dei clienti che hanno un conto corrente in tutti le filiali della banca di Pisa

→ $r(R)$ e $s(S)$ relazioni con $R \sqsubseteq S$, $r \sqsubseteq s$ è una relazione su $R-S$,

una tupla $t \in r \sqsubseteq s$ se

→ $\forall t' \in s, \exists t'' \in r \text{ tale che } t'[S] = t''[S]$
→ $t''[R-S] = t$

Le VISTE

→ Relazioni derivate: contenuto è funzione di altre relazioni ed è possibile che la relazione sia derivata a sua volta a patto che vi sia un ordinamento

↳ Viste materializzate: memorizzate nella base di dati

↳ Relazioni virtuali: definite per mezzo di funzioni non memorizzate all'interno della base di dati, ma utilizzabili come se lo fossero

↳ Hanno il vantaggio che non si creano problemi di inconsistenza dei dati ma devono essere ricalcolate ogni volta per poter essere utilizzate. Quando vengono utilizzate, di fatto il programma le sostituisce con la loro definizione

→ Vantaggi principali

→ Possono essere separati gli accessi a determinati dati introducendo controlli di privacy

→ Espressioni complesse possono essere semplificate, specialmente se presenti espressioni ripetute

→ Un utente può accedere anche solamente a dati significativi

→ In caso di ristrutturazioni, si semplifica il processo

→ Rigidità: Vincoli sulla possibilità di aggiornamento

Metodologie di progettazione

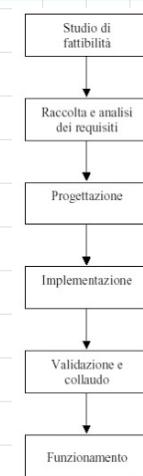
Il ciclo di vita di un sistema

- Studio di fattibilità: costi delle alternative possibili e definizione delle priorità delle cose da realizzare
- Raccolta e analisi dei requisiti: individuazione e studio proprietà e funzionalità del sistema. Prevede l'interazione con l'utente e una descrizione completa ma formale delle specifiche dei dati e delle operazioni. Vengono inoltre stabilite le necessità hardware e software del sistema.
- Progettazione (dei dati e delle applicazioni): Struttura e organizzazione dei dati e dei programmi applicativi. Sono formali e possono procedere in cascata
- Implementazione: realizzazione del sistema con le caratteristiche indicate
- Validazione e collaudo: verifica funzionamento e qualità del sistema in tutte le condizioni operative
- Funzionamento: esecuzione delle operazioni per il quale è stato progettato. Richiede operazioni di gestione e manutenzione

↓

Fasi spesso non sequenziali in quanto c'è la necessità di rivedere azioni precedenti e per questo si ottiene un ciclo. Inoltre si aggiunge la prototipizzazione che prevede l'uso di strumenti software per realizzare una versione semplificata del sistema informativo in modo da testare alcune funzioni

└ Il primo modello definito si chiama Waterfall e prevede azioni eseguite in sequenza per portare alla realizzazione. Quando una fase falliva era necessario ricominciare, data la rigidità del sistema.



La metodologia di progettazione

- Costituita da
 - Decomposizione attività di progetto in passi successivi indipendenti tra loro
 - Strategie da seguire e criteri per la scelta di alternative
 - Modelli di riferimento per descrivere dati in ingresso ed in uscita

Proprietà

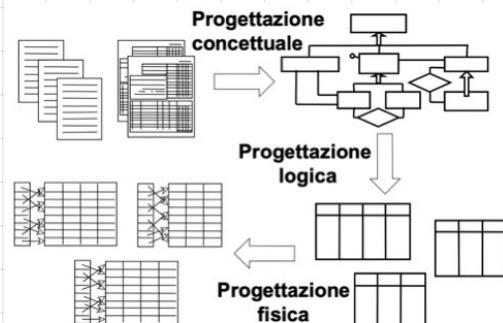
- Generalità rispetto alle applicazioni ad ai sistemi
- Qualità del prodotto
- Facilità di uso



3 Fasi

- Progettazione concettuale: Rappresentazione delle specifiche informali della realtà di interesse in modo formale ed indipendente dal DBMS utilizzato. Le informazioni vengono raccolte dal progettista tramite comunicazione verbale o scritta con gli utenti e tramite documenti preesistenti. Il prodotto di questa fase è lo schema concettuale che prevede un alto livello di astrazione in quanto tiene conto solo dei dati presenti ma non degli aspetti implementativi.
- Progettazione logica: consiste nella traduzione dello schema concettuale nel metodo di rappresentazione dei dati utilizzato dal gestore della base di dati. Grazie allo schema logico si può passare subito alle relazioni. Lo schema logico è ancora indipendente dalla realizzazione fisica. Si compiono inoltre operazioni di ottimizzazione e verifica della qualità, racchiuse all'interno della fase di normalizzazione
- Progettazione fisica: lo schema logico viene completato con le specifiche di memorizzazione dei dati. Il risultato è lo schema fisico e in questo caso dipende dal database stesso

↓



29

I requisiti che una base di dati deve rispettare si dividono in specifiche sui dati e specifiche sulle operazioni. Nella progettazione logica le prime costituiscono lo schema logico, mentre le seconde vengono utilizzate per permettere di eseguire operazioni in modo efficiente. Nella prima fase non è necessario sapere il DBMS utilizzato. Nella fase di progettazione fisica invece è necessario ed inoltre le specifiche sulle operazioni permettono ottimizzazione delle prestazioni del sistema. Il risultato della base di dati è costituito dai 3 schemi, che forniscono una descrizione documentativa, concreta e fisica della base di dati.

Il modello E-R (Entity-Relationship)

- └ Modello utilizzato come standard. È un modello concettuale dei dati e fornisce costrutti che permettono di descrivere la realtà di interesse indipendentemente dal sistema effettivo di realizzazione. I costrutti vengono utilizzati per descrivere schemi che descrivono l'organizzazione e la struttura delle occorrenze dei dati, ovvero dei valori assunti da essi nel tempo. Questo modello prevede la corrispondenza grafica di un elemento a ciascun costrutto. Il linguaggio di programmazione utilizzato in questo caso è il linguaggio UML

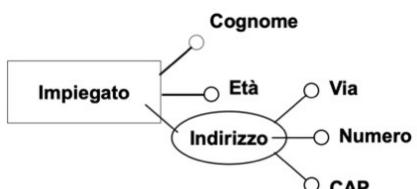
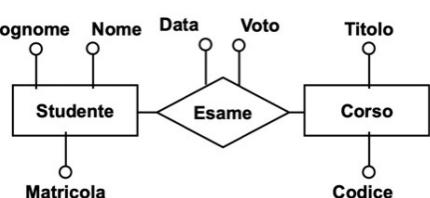
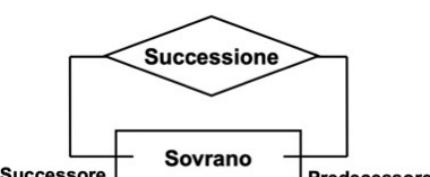
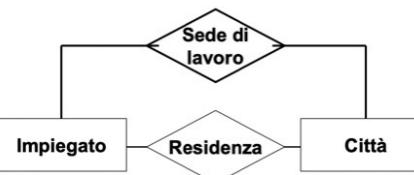
Le entità

- Rappresentano classi di oggetti che hanno proprietà comuni ed esistenza autonoma all'interno della realtà che viene rappresentata
- Un'occorrenza di un'entità costituisce l'oggetto vero e proprio rappresentato dall'entità. Non è quindi un valore che identifica l'oggetto ma bensì l'oggetto in quanto tale. Un'occorrenza quindi esiste indipendentemente dalle proprietà ad essa associate.
- Ha un nome e viene rappresentata con un rettangolo con il nome all'interno. Il nome per convenzione è singolare

Impiegato

Le associazioni / relazioni

- Sono dei legami logici tra due o più entità rilevanti per l'applicazione di interesse
- Un'occorrenza di una relazione è caratterizzata da una ennupla. Nel caso di una relazione tra tre entità si ha quindi una tripla
- Viene identificata univocamente da un nome e rappresentata graficamente da un rombo che contiene il nome all'interno e linee che la collegano alle entità che sono messe in relazione
- Possono esistere relazioni diverse che collegano le stesse entità
- È preferibile utilizzare sostantivi anziché verbi in modo da non assegnare un verso alla relazione
- Essendo relazioni, non possono esserci ennuple ripetute tra le occorrenze di una relazione
- È possibile avere anche relazioni ricorsive, nelle quali è possibile specificare, tramite identificatori associati alle linee uscenti dall'entità, i ruoli dell'entità nella relazioni
- È possibile avere relazioni che coinvolgono più di due entità



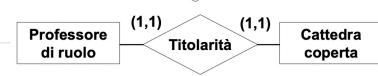
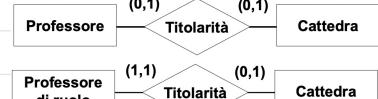
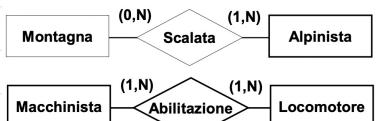
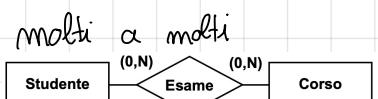
Gli attributi

- Descrivono proprietà elementari di entità e relazioni interessanti ai fini dell'applicazione. Essi associano a ciascuna occorrenza di entità un valore appartenente al dominio, ovvero l'insieme dei valori che può assumere un attributo. Essi non vengono scritti nello schema ma sono riportati nella documentazione associata
- Gli attributi che riguardano proprietà affini nel significato o nell'uso possono essere raggruppati in attributi composti, anche se è preferibile usare attributi atomici
- Vengono identificati dal nome posto sopra ad un pallino collegato all'entità o alla relazione

La cardinalità

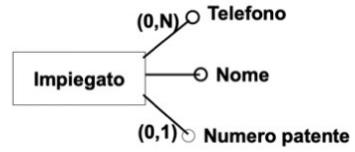
Di relazioni

- Specificare per ciascuna partecipazione di entità a una relazione è descritto il numero minimo e massimo di occorrenze di relazione a cui un'occorrenza di un'entità può partecipare. La cardinalità è descritta da una coppia di numeri, il primo identificante il minimo numero di occorrenze ed il secondo il massimo.
- La cardinalità minima deve essere minore o uguale alla cardinalità massima
- Si utilizzano solo 3 valori: 0, 1, N
- Se la cardinalità minima è 0 la partecipazione dell'entità è opzionale, mentre se è 1 la partecipazione è obbligatoria
- Se la cardinalità massima è 1 si ha una funzione che associa un'occorrenza di un'entità a massimo un'occorrenza dell'altra entità. Se invece è N l'entità può essere associata con un numero arbitrario di entità
- Le cardinalità massime permettono di classificare le relazioni. Se entrambe sono 1, la relazione si dice uno a uno. Se una è 1 e l'altra N si ha una relazione uno a molti. Se entrambe sono N si ha una relazione molti a molti
- La partecipazione obbligatoria è rara in quanto non sono note le occorrenze delle entità collegate
- Se si ha una relazione n-aria ed un'entità partecipa con cardinalità massima 1 è possibile legare l'entità con le altre sfruttando una relazione binaria molti a molti



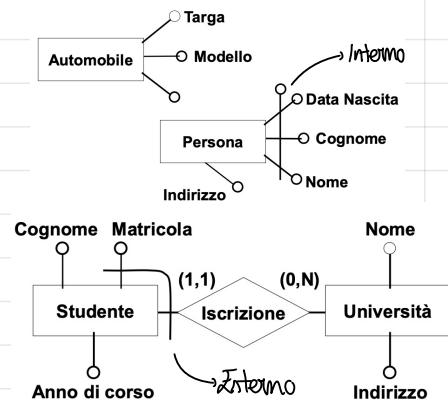
Di attributi

- Descrivono il numero minimo e massimo di valori dell'attributo per ogni occorrenza di entità o relazione
- Se è (1,1) viene omessa e si ha una funzione che associa ogni occorrenza ad un solo valore dell'attributo.
- L'attributo si dice obbligatorio
- Se è 0 un attributo si dice opzionale e in questo caso il valore può mancare
- Se è N l'attributo può assumere più valori e si dice multivaleure. È preferibile sostituirli con entità



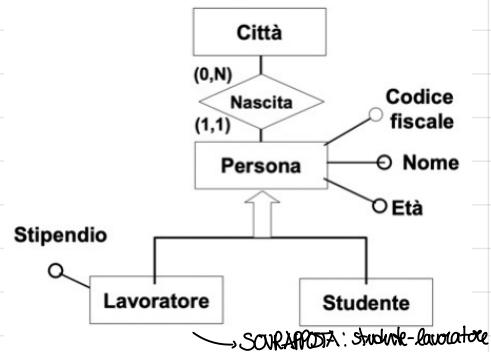
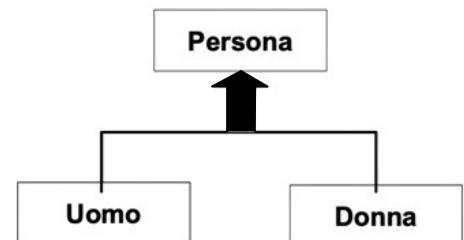
Gli identificatori delle entità

- Permettono di identificare univocamente le occorrenze delle entità.
- Se l'identificatore è costituito da attributi dell'entità si ha un identificatore interno (chiave)
- Se quello interno non è sufficiente si ricorre all'identificatore esterno, dato da un attributo di un'altra entità unito ad un'altra entità. Questo è possibile se le due entità sono messe in relazione da una relazione uno ad uno.
- Gli attributi identificatori devono avere cardinalità (1,1)
- Non è possibile fare ciclo di identificatori esterni
- Ogni entità deve avere un identificatore ma può averne più di uno. Nel caso di più identificatori gli attributi possono essere opzionali



Le generalizzazioni

- Legami logici tra E, detta entità genitore, ed E1...En, dette entità figlie. E è la più generale, che comprende le figlie come casi particolari. E è quindi generalizzazione di E1...En ed E1...En sono specializzazioni di E
- Ogni occorrenza di un'entità figlia è occorrenza dell'entità genitore
- Ogni proprietà dell'entità genitore è posseduta dalle entità figlie (ereditarietà)
- Una generalizzazione è totale se ogni occorrenza dell'entità genitore è occorrenza di almeno una delle figlie, altrimenti è parziale. Si indica con una freccia piena se è totale, con una freccia vuota se parziale.
- Se è parziale è necessario che l'identificatore sia posizionato sull'entità genitore
- Una generalizzazione è esclusiva se ogni occorrenza dell'entità genitore è al più un'occorrenza di una delle unità figlie, è sovrapposta altrimenti. Le generalizzazioni sovrapposte possono essere trasformate in esclusive aggiungendo entità che rappresentano i casi di sovrapposizione
- Un'entità può essere coinvolta in più generalizzazioni
- Possono esserci gerarchie di generalizzazioni
- Una generalizzazione può avere una sola entità figlia: sottoinsieme



Documentazione di schemi ER

- Di supporto: facilita l'interpretazione dello schema e permette di rappresentare dati che non possono essere espressi tramite i costrutti del modello
- Regole aziendali: informazioni che definiscono o vincolano qualche aspetto di un'applicazione
 - Descrizione di un concetto rilevante per l'applicazione: linguaggio naturale sotto forma di glossario
 - Vincolo di integrità: definizioni formali ma senza una standardizzazione. Vengono utilizzate asserzioni, ovvero affermazioni che devono essere sempre verificate nella base di dati. Devono essere atomiche, ovvero non scomponibili e devono essere enunciate in maniera dichiarativa, ovvero nella forma: <conceitto> deve/non deve <espressione su concetti>
 - Derivazione: specificano le operazioni che permettono di ottenere il concetto derivato. Sono identificate da RD ed un numero progressivo e sono nella forma: <conceitto> si ottiene <operazione su concetti>
- Dizionario dei dati: due tabelle. La prima descrive le entità dello schema con nome, definizione informale, elenco di attributi e l'identificatore. La seconda descrive le relazioni con nome, descrizione informale, elenco di attributi ed elenco di entità associate alla loro cardinalità. Per le regole aziendali è possibile usare un'altra tabella specificando il tipo di regole espresse

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

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

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

Regole di derivazione
(1) Il numero di impiegati di un dipartimento si ottiene contando gli impiegati che afferiscono a tale dipartimento
(2) Il budget di un progetto si ottiene moltiplicando per 3 la somma degli stipendi degli impiegati che vi partecipano

La progettazione concettuale

La raccolta dei requisiti

→ Individuazione dei problemi che l'applicazione deve risolvere e della struttura dei dati e delle operazioni. I requisiti vengono inizialmente raccolti in linguaggio naturale e in modo disorganizzato e poi verranno analizzati e riorganizzati in seguito.

I requisiti provengono generalmente da

- Utenti dell'applicazione: raccolti tramite interviste anche ripetute
- Documentazione esistente
- Realizzazioni preesistenti: applicazioni che devono essere sostituite o che devono interagire con il nuovo sistema

↓
Utenti diversi possono fornire informazioni diverse e talvolta contraddittorie. È necessario quindi verificare la correttezza facendosi fornire esempi oppure definizioni e classificazioni precise, in modo da poter raffinare l'informazione raccolta. È inoltre necessaria una precisa analisi del linguaggio per prevenire ambiguità derivanti dal linguaggio naturale.

→ Regole generali

- Scegliere il corretto livello di astrazione: non utilizzare termini né troppo generici né troppo specifici
- Standardizzare la struttura delle frasi
- Evitare frasi contorte
- Individuare sinonimi ed omonimi ed unificare i termini: creano ambiguità e per questo nel caso di sinonimi è consigliato unire i termini e nel caso di omonimi differenziarli
- Rendere esplicito il collegamento tra i termini: l'assenza di un contesto rende alcuni concetti ambigu
- Costruire un glossario dei termini: contiene descrizione, sinonimi

Oltre a queste, viene decomposto il testo in frasi omogenee, ovvero relative agli stessi concetti e vengono anche raccolte le specifiche sulle operazioni, indicando anche la frequenza con la quale vengono svolte, utilizzando la stessa terminologia usata per i dati.

Criteri generali di rappresentazione

- Se un concetto ha proprietà significative e/o descrive classi di oggetti con esistenza autonoma è opportuno utilizzare un'entità
- Se un concetto ha una struttura semplice e non possiede proprietà rilevanti è opportuno porlo come attributo del concetto a cui si riferisce
- Se sono state individuate due o più entità che presentano una relazione logica tra di loro è opportuno collegarle con una relazione
- Se uno o più concetti sono casi particolari di un altro è opportuno utilizzare una generalizzazione

Le strategie di progetto

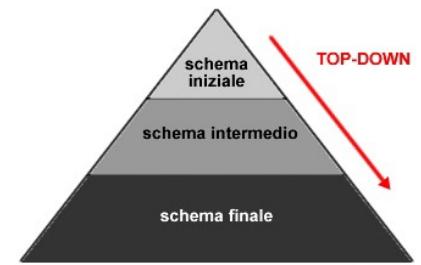
La strategia top-down

→ Prevede una serie di raffinamenti successivi a partire da uno schema che descrive tutte le specifiche con pochi concetti molto astratti. Lo schema viene raffinato aumentando il dettaglio dei concetti in esso descritti. Ogni raffinamento produce uno schema con un diverso (maggiore) livello di dettaglio

→ Primitive di trasformazione

- Definizione di attributi per un'entità o per una relazione
- Reificazione di un attributo o di una entità
- Decomposizione di una relazione in due distinte
- Trasformazione di entità in una generalizzazione

→ Vantaggio: possibile descrivere inizialmente entità senza specificare troppi dettagli ma questo non è possibile almeno che non si abbia una conoscenza astratta di tutte le componenti del sistema



La strategia bottom-up

→ Le specifiche vengono decomposte in schemi elementari (possono essere costituiti anche da una sola entità) che vengono poi fusi insieme per rappresentare l'informazione completa. I concetti presenti vengono introdotti gradualmente durante la progettazione del database.

→ Primitive di trasformazione

- Introduzione di nuove entità o relazioni
- Individuazione di legami riconducibili a generalizzazioni
- Aggregazione di attributi in entità e relazioni

→ Vantaggio: la decomposizione in componenti semplici permette una realizzazione di gruppo del progetto più semplice

→ Svantaggio: integrazione di schemi diversi è difficile

La strategia inside-out

→ È un caso particolare della metodologia bottom up in cui viene identificata un'entità semplice dalla quale si procede a macchia d'olio, ovvero sviluppando prima i concetti in relazione con le entità iniziali

→ Il vantaggio è quello di non prevedere passi di integrazione, ma lo svantaggio è la necessità dell'analisi delle specifiche per individuare concetti non rappresentati e descriverli in dettaglio

La strategia mista

Combina i vantaggi della top down a quelli della bottom up in quanto i requisiti vengono divisi in entità separate ed allo stesso tempo viene definito uno schema scheletro contenente i concetti principali che favorisce l'integrazione degli schemi sviluppati separatamente. È la più flessibile ed integra anche la strategia inside out ed è quella utilizzabile nella maggior parte dei casi.

Le qualità di uno schema concettuale

- **Correttezza:** uso corretto dei costrutti del linguaggio. Possono esserci sia errori sintattici, riguardanti l'uso errato dei costrutti, che semantici, riguardanti un uso errato dei costrutti in base alla loro definizione. La correttezza si verifica con il confronto dello schema con le specifiche.
- **Completezza:** rappresenta tutti i dati e le operazioni di interesse e si verifica accertandosi della presenza dei dati delle specifiche nello schema.
- **Leggibilità:** rappresenta i requisiti in maniera naturale e comprensibile. Per far questo si utilizza una griglia con al centro gli elementi che possiedono più relazioni, si tracciano linee perpendicolari evitando intersezioni e si dispongono le entità padri delle generalizzazioni sopra le figlie.
- **Minimalità:** non devono essere presenti ridondanze e se questo sono presenti vanno documentate e devono essere strettamente necessarie.

Metodologia generale

- **Analisi dei requisiti:** costruzione del glossario, analisi dei requisiti ed eliminazione delle ambiguità e raggruppare i requisiti omogenei.
- **Passo base:** individuare i concetti fondamentali e costruire uno schema scheletro.
- **Passo di decomposizione:** decomposizione dei requisiti con riferimento ai concetti presenti nello schema scheletro.
- **Passo iterativo:** raffinamenti dei concetti presenti e aggiunta di nuovi.
- **Passo di integrazione:** integrazione di sottoschemi in unico schema principale.
- **Analisi di qualità:** verificare la correttezza, completezza, leggibilità e minimalità dello schema ed eventualmente ristrutturararlo.

Da progettazione logica

Le fasi della progettazione logica

- Ristrutturazione dello schema E-R: indipendente dal modello logico e basata su criteri di ottimizzazione dello schema e di semplificazione della fase successiva.
- Prende in ingresso lo schema concettuale prodotto dalla fase precedente ed il carico applicativo previsto e restituisce uno schema ristrutturato che tiene conto degli aspetti realizzativi
- Traduzione verso il modello logico: fa riferimento ad un modello logico preciso e si basa sulle sue caratteristiche. Prende in ingresso lo schema ristrutturato che viene certificato ed ottimizzato. Produce lo schema logico, i vincoli di integrità e la documentazione associata

Analisi delle prestazioni di schemi E-R

- Costo di un'operazione: numero di occorrenze di entità ed associazioni che vanno visitate per rispondere ad un'operazione sul database
- Occupazione di memoria: spazio di memoria utilizzato per memorizzare i dati descritti dallo schema
- Volume dei dati: numero di occorrenze di entità ed associazioni e dimensioni degli attributi
- Caratteristiche delle operazioni: tipo dell'operazione, frequenza e dati coinvolti



La tavola dei volumi riporta i concetti dello schema con il volume previsto a regime che dipende dal numero di occorrenze dell'entità e dal numero medio di partecipazione delle stesse alle associazioni, mentre la tavola delle operazioni riporta la frequenza e specifica con I se l'operazione è interattiva e con B se invece è batch. È possibile inoltre descrivere graficamente i dati coinvolti con uno schema di operazione che è un sottoinsieme dello schema E-R sul quale viene riportato il cammino logico fatto per accedere alle informazioni di interesse

La stima del costo di un'operazione dipende da

- Numero di accessi al primo concetto presente nel cammino
- prosecuzione nel cammino



Scritto nella tavola degli accessi: si distingono operazioni di lettura e scrittura in quanto le ultime sono più onerose. Un'operazione in scrittura costa il doppio

Da ristrutturazione: Analisi delle redundanze

- 4 forme:
 - Attributi derivabili da attributi della stessa entità
 - Attributi derivabili da attributi di altre entità
 - Attributi derivabili da operazioni di conteggio di occorrenze
 - Associazioni derivabili dalla composizione di altre associazioni in presenza di cicli

Il vantaggio è la riduzione degli accessi necessari per calcolare l'attributo, mentre lo svantaggio è costituito dal costo di tenere aggiornati anche i dati derivati. Nella ristrutturazione è quindi utile tenere di conto di entrambi questi fattori

Da ristrutturazione: eliminazione delle generalizzazioni

- 3 possibilità:
 - Accorpamento delle figlie della generalizzazione nel genitore: le entità figlie vengono eliminate e i loro attributi vengono passati al padre con l'aggiunta di un nuovo attributo che permette di identificare a che occorrenze del figlio deve appartenere il padre. Nel caso di una generalizzazione non totale l'attributo può essere opzionale
 - Accorpamento del genitore della generalizzazione nelle figlie: il padre viene eliminato e tutte le sue proprietà, comprese le relazioni, vengono trasferite ai figli
 - Sostituzione della generalizzazione con associazioni: si trasforma in due associazioni uno ad uno che legano l'entità genitore con i figli senza trasferire attributi ed associazioni. L'entità padre non può partecipare contemporaneamente ad entrambe le relazioni che la legano ai figli e se la generalizzazione è totale deve esserci almeno una partecipazione
- Come scegliere?
 - 1) operazioni non fanno distinzione tra genitore e figli in quanto ad occorrenze ed attributi. C'è uno spreco di memoria di valori nulli che però assicura minori accessi in memoria
 - 2) possibile solo se la generalizzazione è totale ed è conveniente se è fatta distinzione tra le entità figlie
 - 3) conveniente se non è totale e se vengono fatti accessi separati a padre e figli. C'è un incremento di accessi per assicurare la consistenza delle occorrenze a causa dei vincoli introdotti

Da ristrutturazione: partizionamento / accorpamento

- Partizionamento di entità: conveniente se vengono spesso utilizzate determinate porzioni di attributi di un'entità. È possibile partizionare verticalmente, ed in questo caso si opera sugli attributi, oppure orizzontalmente, se si lavora sulle occorrenze delle entità. Le decomposizioni orizzontali sono in pratica delle generalizzazioni ed hanno l'effetto collaterale di duplicare tutte le associazioni dell'entità partizionata. Le partizioni verticali producono invece entità con pochi attributi che però forniscono una grande quantità di informazione
- Eliminazione di attributi multivalue: Eliminati in quanto non rappresentabili e sostituiti da entità ed associazioni
- Accorpamento di entità: fatto se sono utilizzati frequentemente dati appartenenti a più entità collegate da un'associazione per risparmiare accessi. Un effetto collaterale è la possibile presenza di valori nulli
- Altri tipi: è possibile dividere un'associazione tra due o più entità in più associazioni per separare occorrenze dell'associazione originale accedute separatamente

La ristrutturazione: scelta degli identificatori principali

Le chiavi permettono di stabilire legami tra dati in relazioni diverse e il linguaggio prevede di specificare una chiave primaria per costruire gli indici in modo da poter accedere in modo efficiente ai dati.

Criteri per decidere quale identificatore diventa una chiave

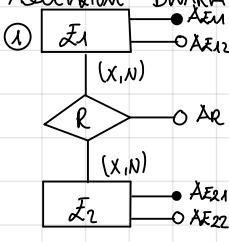
- Attributi con valori nulli non possono essere chiavi primarie perché non permettono l'accesso a tutte le occorrenze dell'entità corrispondente
- È preferibile utilizzare il minor numero di attributi possibile per aumentare l'efficienza
- Un identificatore interno è preferibile ad uno esterno perché nell'ultimo caso si generano chiavi con molti attributi
- È preferibile utilizzare un identificatore utilizzato spesso per trarre vantaggio dagli indici generati dal DBMS

Se nessun identificatore soddisfa queste caratteristiche viene introdotto il codice. È conveniente lasciar indicati anche gli identificatori secondari perché utili in fase di progettazione fisica

Trascrizione verso il modello relazionale

Entità e associazioni molti a molti → Invece i vincoli di integrità referenziale dall'algoritmo

ASSOCIAZIONE BINARIA

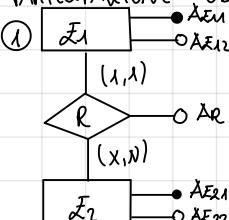


→ Per ogni entità deve essere presente una relazione con lo stesso nome con gli stessi attributi dell'entità e il suo identificatore come chiave. L'associazione è invece data da una relazione con gli attributi dell'associazione stessa e gli identificatori delle entità coinvolte. Se gli attributi di entità e associazioni sono opzionali, i corrispondenti attributi qui rappresentati possono avere valori nulli

- ① $E_1(A_{E11}, A_{E12})$
 $E_2(A_{E21}, A_{E22})$
 $R(A_{E11}, A_{E21}, A_R)$
- ② $E_1(A_{E11}, A_{E12})$
 $E_2(A_{E21}, A_{E22})$
 $E_3(A_{E31}, A_{E32})$
 $R(A_{E11}, A_{E21}, A_{E31}, A_R)$

Associazioni uno a molti

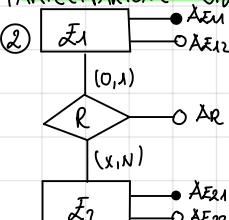
PARTECIPAZIONE OBBLIGATORIA



→ Le relazioni E1 ed R hanno la stessa chiave. È quindi preferibile fonderle in un'unica relazione E1 che rappresenta sia E1 che R.

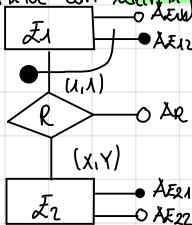
① $E_1(A_{E11}, A_{E12}, A_{E21}, A_R)$ → Presenti le chiavi di E1 ed E2
 $E_2(A_{E21}, A_{E22})$

PARTECIPAZIONE OPZIONALE



→ ② $E_1(A_{E11}, A_{E12}, A_{E21}, A_R^*)$
 $E_2(A_{E21}, A_{E22})$
③ $E_1(A_{E11}, A_{E12})$
 $E_2(A_{E21}, A_{E22})$ → È possibile la presenza di valori nulli e per questo non era adatta al primo caso
 $R(A_{E11}, A_{E21}, A_R)$

Entità con identificatore esterno

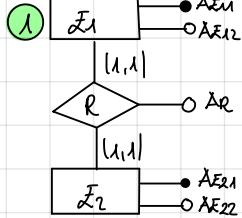


→ Danno luogo a relazioni che includono gli identificatori delle entità identificate. È sufficiente infatti rappresentare l'identificatore esterno per rappresentare l'associazione in quanto entrambe le entità partecipano alla relazione con cardinalità minima e massima uguale a 1

④ $E_1(A_{E12}, A_{E21}, A_{E11}, A_R)$
 $E_2(A_{E21}, A_{E22})$

● Associazioni uno a uno

PARTECIPAZIONE OBBLIGATORIA



- 1.1 E1 (AE11, AE12, AE21, AR)
 E2 (AE21, AE22)
- 1.2 E2 (AE11, AE22, AE21, AR)
 E1 (AE11, AE12)

ATT: Notazioni

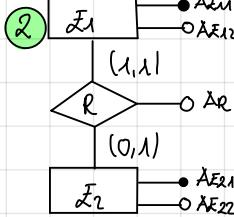
X, Y: qualsunque cardinalità

*: attributo opzionale

--: chiave secondaria

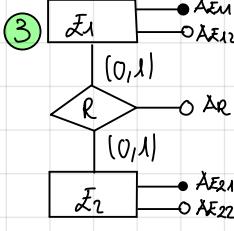
—: chiave primaria

PARTECIPAZIONE OPZIONALE PER UN'ENTITÀ



- 2 E1 (AE11, AE12, AE21, AR)
 E2 (AE21, AE22)

PARTECIPAZIONE OPZIONALE PER ENTRAMBE LE ENTITÀ



- 3.1 E1 (AE11, AE12)
 E2 (AE21, AE22, AE11, AR)
- 3.2 E1 (AE11, AE12, AE21, AE22)
 E2 (AE21, AE22)
- 3.3 E1 (AE11, AE12)
 E2 (AE21, AE22)
 R (AE11, AE12, AR)

● Documentazione di schemi logici

Gran parte della documentazione ricevuta in ingresso all'inizio della progettazione logica è riutilizzabile anche come risultato della stessa. Inoltre, se i nomi dei concetti usati per creare lo schema ER sono rimasti invariati, può essere riutilizzata anche la documentazione riguardante le regole aziendali. È necessario però descrivere anche i vincoli di integrità e a questo scopo vengono utilizzati diagrammi dove le frecce rappresentano questi vincoli, le chiavi sono evidenziate in grassetto e gli asteriski sugli attributi indicano che questi sono opzionali.

Dipendenze funzionali e normalizzazione

• Regole generali per uno schema corretto

→ Una volta progettato lo schema concettuale è necessario **migliorarlo** facendo alcuni aggiornamenti con due obiettivi principali

→ Conservare l'informazione, cioè mantenere **tutti i concetti rappresentati** tramite il modello concettuale

→ Minimizzare la ridondanza in modo da evitare di memorizzare più volte la stessa informazione, il che comporta in genere la necessità di aggiornarla molteplici volte

→ Esistono fondamentalmente **3 linee guida principali** per un buon progetto

→ Regola 1: Lo schema deve essere **semplice da spiegare**. Non devono quindi esserci **anomalie semantiche** e per questo devono essere accostate solo entità dello stesso tipo ed attributi che si riferiscono alle stesse entità. Un **diagramma ordinato** permette di **capire meglio il progetto** e le **proprietà del database**

→ Regola 2: Gli schemi **non devono presentare anomalie di modifica, cancellazione o inserimento**. Le anomalie di aggiornamento riguardano principalmente i dati **ridondanti**. Essi infatti necessitano aggiornamenti multipli, che devono riguardare tutte le istanze degli attributi ripetuti. Le anomalie **di cancellazione** si presentano invece quando una tupla rappresenta **concetti eterogenei**. In questo caso la cancellazione della tupla porta alla **cancellazione di tutti i concetti da essa espressi**, anche di quelli che potrebbero conservare la loro validità. Le anomalie **d'inserimento** si hanno sempre in presenza di una relazione che rappresenta concetti eterogenei perché anche in questo caso è **impossibile inserire concetti riguardanti soltanto una parte della relazione**. La mancanza delle anomalie deve essere **certificata** utilizzando una **descrizione formale** dei fatti descritti in uno schema relazionale. Se sono presenti anomalie è necessario **rilevarle** ed assicurarsi che la **base di dati e le applicazioni** che vi operano **funkzionino correttamente**.

→ Regola 3: E' meglio **evitare** la presenza di relazioni i cui **valori possono essere frequentemente nulli**. Se presenti, devono esserci soltanto in casi eccezionali e comunque in numero minore rispetto alle occorrenze di una relazione.

Esempio: FATTURA (CodF, CodProd, TotDaPagare, CostoNettoProd, IVA)

Semantica: -CodF determina CodProd e TotDaPagare

-CodProd determina CostoNettoProd e IVA

-CostoNettoProd e IVA determinano TotDaPagare

Attenzione: -TotDaPagare deve essere consistente con la regola che lo lega a CostoNettoProd e IVA

-A CodProd deve essere attribuita la giusta percentuale di IVA: Se l'IVA cambia è necessario modificare anche TotDaPagare e quindi ai fini della consistenza del database è bene che questo attributo non sia presente

• Le dipendenze funzionali

→ Per evitare le anomalie che abbiamo citato sopra, sono presenti delle **forme normali**. Ciascuna di esse garantisce l'**assenza di determinati difetti** in uno schema di relazione R e quindi ne garantisce un certo livello di qualità. Esistono dei test per verificare che uno schema soddisfa le proprietà di una generalizzazione. La **normalizzazione** è il processo che ci permette di trasformare uno schema in una determinata forma normale. Le forme normali non devono essere quindi viste come delle metodologie di progettazione ma sono bensì tecniche di verifica di risultati che permettono di migliorare uno schema una volta che esso è stato progettato.

→ Le **dipendenze funzionali** esprimono **vincoli di integrità** per il **modello relazionale** che descrivono **legami di tipo funzionale tra gli attributi di una relazione**. In particolare esse esprimono un **legame semantico** tra **due gruppi di attributi** di uno **schema di una relazione R** e si aggiungono ai vincoli di **chiave primaria**. Le dipendenze funzionali sono **proprietà generali** di una relazione R e non di un suo stato valido r. Per questo, esse non possono essere dedotte da r ma devono bensì essere **specificate** da qualcuno che conosce le proprietà degli attributi di R.

→ Possiamo quindi fornire una definizione formale di dipendenza funzionale. Si considerino la relazione r su R(X) e due sottoinsiemi non vuoti Y e Z di X. Esiste in r una dipendenza funzionale FD da Y a Z se, per ogni coppia di tuple t1 e t2 di r con gli stessi valori su Y, risulta che t1 e t2 hanno gli stessi valori anche su Z. La notazione utilizzata è $Y \rightarrow Z$. La cosa importante da notare in questo caso è la **direzionalità della dipendenza**. Non è detto infatti che esista la dipendenza funzionale nel verso opposto. Un'altra caratteristica importante delle dipendenze funzionali è la seguente: se Z è composto da una serie di attributi, la dipendenza funzionale deve essere **soddisfatta su tutti gli attributi che compongono Z**. Se Z per esempio è composto da A1, A2, ..., Ak, allora la dipendenza funzionale $Y \rightarrow A$ significa che essa è soddisfatta su tutti gli attributi di Z.

→ Parliamo adesso di alcune tipologie di dipendenze funzionali:

→ Se è verificata $Y \rightarrow Z$ e si ha anche che $W = Y$, allora la dipendenza funzionale si dice **completa** se non vale $W \rightarrow Z$

→ Se Y è una superchiave di R(X), allora Y determina ogni altro attributo della relazione $Y \rightarrow X$. Si ha inoltre che essendo semplicemente una superchiave e non una chiave primaria, allora la relazione può non essere completa in quanto possono essere rimossi attributi. E' invece completa se la relazione riguarda una chiave primaria. Il vincolo di chiave è una dipendenza funzionale proprio per la definizione di chiave, in quanto non possono esistere due tuple con gli stessi valori della relazione. Una dipendenza funzionale con una chiave al primo membro è quindi sempre verificata. Possiamo quindi dire che una dipendenza funzionale generalizza il vincolo di chiave. Una dipendenza funzionale $Y \rightarrow Z$ su uno schema R(X) in particolare degenera in un vincolo di chiave primaria l'unione di Y e Z è uguale a X.

→ Le dipendenze funzionali di tipo $Y \rightarrow A$ sono banali se A appartiene a Y, mentre non è banale se non vi appartiene oppure, se è di tipo $Y \rightarrow Z$, nessun attributo in Z appartiene a Y

Impiegato	Stipendio	Progetto	Bilancio	Funzione
Rossi	20	Marte	2	tecnico
Verdi	35	Giove	15	progettista
Verdi	35	Venere	15	progettista
Neri	55	Venere	15	direttore
Neri	55	Giove	15	consulente
Neri	55	Marte	2	consulente
Mori	48	Marte	2	direttore
Mori	48	Venere	15	progettista
Bianchi	48	Venere	15	progettista
Bianchi	48	Giove	15	direttore

Dipendenze funzionali:

Impiegato → Stipendio

Progetto → Bilancio

Impiegato Progetto → Funzione

Impiegato Progetto → Progetto: Progetto è da entrambe le parti, quindi la dipendenza funzionale è banale.

Legame con le anomalie: Essendo che Impiegato Progetto è una chiave, avendo Impiegato Progetto → Funzione non ci sono ripetizioni e quindi si elimina un'anomalia

L'implicazione, la chiusura e la superchiave

→ Una caratteristica fondamentale delle dipendenze funzionali è l'**implicazione**. Si dice che un insieme di dipendenze funzionali F implica un'altra dipendenza f se ogni relazione che soddisfa tutte le dipendenze in F soddisfa anche f .

In particolare, una definizione formale può essere la seguente: sia F un insieme di dipendenze funzionali definite su $R(Z)$ e sia $X \rightarrow Y$: si dice che F implica $X \rightarrow Y$ ($F \rightarrow X \rightarrow Y$) se, per ogni istanza r di R che verifica tutte le dipendenze in F , risulta verificata anche $X \rightarrow Y$.

Si dice anche che $X \rightarrow Y$ è implicata da F .

→ Un concetto direttamente legato all'implicazione è la **chiusura**. Una definizione formale di chiusura è la seguente: Dato un insieme di dipendenze funzionali F definite su $R(Z)$, la chiusura di F è l'insieme di tutte le dipendenze funzionali implicate da F e si indica con F^+ . In simboli, $F^+ = \{X \rightarrow Y \mid F \rightarrow X \rightarrow Y\}$. Dato un insieme di dipendenze funzionali F definite su $R(Z)$, un'istanza r di R che soddisfa F soddisfa anche F^+ .

→ Un altro concetto importante per quanto riguarda l'implicazione è la **superchiave**. Dato $R(Z)$ ed un insieme F di FD, un insieme di attributi di K appartenenti a Z , si dice superchiave di R , se la dipendenza funzionale $K \rightarrow Z$ è logicamente implicata da F ($K \rightarrow Z$ in F^+). Se nessun sottoinsieme proprio di K è superchiave di R , allora K si dice chiave di R . Nel caso di chiave primaria viene scelto l'**identificatore primario**, ovvero quello che nelle tabelle viene utilizzato come primary key. Per trovare la chiave si utilizza un algoritmo di complessità esponenziale. L'algoritmo analizza le dipendenze e lavora nel modo seguente. Se un attributo sta solo a sinistra delle dipendenze, allora non c'è modo di ottenerne quell'attributo e quindi fa parte della chiave. Questi attributi li poniamo nell'insieme N . Se invece ci sono attributi che sono presenti soltanto a destra, essi non fanno parte della chiave in quanto non portano da nessuna parte in termini di direzione. Man mano che vengono aggiunti attributi all'insieme N si ampliano sempre di più le direzioni che possono essere raggiunte con le dipendenze funzionali. Il tempo impiegato è esponenziale perché devono essere analizzati tutti i sottoinsiemi. Lo stesso algoritmo può essere utilizzato anche per trovare F^+ .

Le regole di Armstrong

→ La definizione di F^+ si presta male all'applicazione pratica a causa della presenza di per ogni nella definizione. Per questo, per calcolare F^+ si utilizzano delle regole, dette **regole di inferenza di Armstrong**, che permettono di derivare costruttivamente tutte le dipendenze funzionali che sono implicate da un insieme iniziale. Tali regole sono corrette e complete, in quanto grazie a queste è possibile costruire tutto l'insieme chiusura. Tali regole sono anche minimali. Le prime 3 regole principali sono seguenti:

→ **Riflessività:** Se $Y \subseteq X$, allora $X \rightarrow Y$

→ **Additività:** Se $X \rightarrow Y$, allora $XZ \rightarrow YZ$, per qualunque Z

Dimostrazione: Supponiamo per assurdo che esista una istanza r di R in cui valga $X \rightarrow Y$ ma non $XZ \rightarrow YZ$, devono perciò esistere due tuple t_1 e t_2 di r tali che :

$$(1) t_1[X] = t_2[X], (2) t_1[Y] = t_2[Y], (3) t_1[XZ] = t_2[XZ], (4) t_1[YZ] \neq t_2[YZ]$$

ma ciò è assurdo, poiché da (1) e (3) si deduce: (5) $t_1[Z] = t_2[Z]$, e da (2) e (5) si deduce: (6) $t_1[YZ] = t_2[YZ]$, in contraddizione con la (4).

→ **Transitività:** Se $X \rightarrow Y$ e $Y \rightarrow Z$, allora $X \rightarrow Z$

Dimostrazione: Supponiamo per assurdo che esista una istanza r di R in cui valgano $X \rightarrow Y$ e $Y \rightarrow Z$, ma non $X \rightarrow Z$, devono perciò esistere due tuple t_1 e t_2 in r tali che :

$$(1) t_1[X] = t_2[X], (2) t_1[Y] = t_2[Y], (3) t_1[Z] = t_2[Z], (4) t_1[Z] \neq t_2[Z] \text{ ma ciò è assurdo.}$$

→ Esistono 3 teoremi riguardanti le proprietà di Armstrong:

→ Teorema 1: Le regole di inferenza di Armstrong sono corrette, cioè, applicandole ad un insieme F di dipendenze funzionali, si ottengono solo dipendenze logicamente implicate da F .

→ Teorema 2: Le regole di inferenza di Armstrong sono complete, cioè, applicandole ad un insieme F di dipendenze funzionali, si ottengono tutte le dipendenze logicamente implicate da F .

→ Teorema 3: Le regole di inferenza di Armstrong sono minimali, cioè ignorando anche una sola di esse, l'insieme delle regole che rimane non è più completo.

→ L'algoritmo per trovare la chiusura opera nel modo seguente: per ciascuna dipendenza si applica la riflessività finché è possibile e successivamente l'arricchimento. Le dipendenze che si ottengono vengono aggiunte all'insieme chiusura. Dopodiché per ogni coppia dell'insieme attuale, viene verificata la transitività, determina quali coppie di dipendenze entreranno a far parte di F^+ . Questo verrà applicato fin quando si verifica che F^+ rimane invariato.

In modo più specifico, l'algoritmo lavora nel modo seguente:

```
F+ = F
repeat
    for each f ∈ F+ do
        applica riflessività ed arricchimento ad f
        aggiungi ad F+ le dipendenze ottenute
    end for;
    for each f1, f2 ∈ F+ do
        if f1 ed f2 possono essere combinate usando la transitività'
            then aggiungi ad F+ le dipendenze ottenute
        end if
    end for
until F+ non cambia
```

Vi sono anche altre proprietà derivabili dalle precedenti:

- Regola di unione: $\{X \rightarrow Y, X \rightarrow Z\} \rightarrow X \rightarrow YZ$
- Regola di pseudotransitività: $\{X \rightarrow Y, WY \rightarrow Z\} \rightarrow XW \rightarrow Z$
- Regola di decomposizione: Se $X \subseteq Y, X \rightarrow Y \rightarrow X \rightarrow Z$

Esempio: $R = (A, B, C, G, H, I)$ $F = \{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$

Membri di F^+ :
- $A \rightarrow H$ per la proprietà di transitività da $A \rightarrow B$ e $B \rightarrow H$
- $AG \rightarrow I$ arricchendo $A \rightarrow C$ con G e poi utilizzando la transitività con $CG \rightarrow I$
- $CG \rightarrow HI$ arricchendo $CG \rightarrow I$ con CG per ottenere $CG \rightarrow ICG$ e poi arricchimento di $CG \rightarrow H$ con I per ottenere $CGI \rightarrow H$ e poi transitività

L'equivalenza

- Una caratteristica importante da utilizzare quando si maneggiano dipendenze funzionali è l'**equivalenza**. Essa dà la possibilità di sostituire un insieme di dipendenze funzionali con un altro che rispetta le stesse proprietà ma che è più semplice. La definizione formale è la seguente: F e G sono equivalenti se $F^+ = G^+$, ovvero, per ogni $X \rightarrow Y \subseteq F$, deve essere $X \rightarrow Y \in G^+$ e, viceversa, per ogni $X \rightarrow Y \subseteq G$, deve essere $X \rightarrow Y \in F^+$. In pratica si ha che due insiemi sono equivalenti se ciascuna dipendenza di un insieme ne implica una dell'altro insieme e viceversa. Quando due insiemi sono equivalenti si dice che uno è copertura dell'altro.
- Una conseguenza importante dell'equivalenza è il seguente teorema: $X \rightarrow Y$ è nella chiusura F^+ se Y è contenuto nella chiusura di X . Questo teorema semplifica di molto il calcolo di F^+ in quanto banalmente esso non è più necessario. Infatti, in molti casi è necessario verificare che F^+ contenga una certa dipendenza invece che determinare tutta F^+ . Calcolare X^+ è semplice e si fa con il seguente algoritmo:

CalcolaChiusura(X, F) =

```
{  $X^+ = X;$ 
repeat
    for each  $Vi \rightarrow Wi$  in  $F$  do
        if  $Vi \subseteq X^+$  e  $Wi \not\subseteq X^+$  then  $\{X^+ = X^+ \cup Wi\}$ 
    end if
end for
until  $X^+$  non cambia
}
```

Per ogni dipendenza che esiste in F , se la parte sinistra è contenuta in X^+ , mentre non lo è contenuta la parte destra allora aggiungo la parte destra, altrimenti non la aggiungo. La complessità di questo algoritmo non è più esponenziale perché, aggiungendo un attributo per passaggio, al massimo si aggiungono n attributi. Per quanto riguarda gli attributi è lineare (n), mentre per quanto riguarda attributi e confronti (n^2n) si ha n^2 .

→ Un utilizzo importante del teorema precedente si ha nel trovare la superchiave: X è superchiave di R se e solo se $X \rightarrow Z$ è in F^+ , cioè per il teorema precedente $Z \subseteq X^+$.

Le ridondanze

- Il concetto di equivalenza tra insieme può essere utilizzato per ridurre le dipendenze in un certo insieme, che possono essere ridondanti. Cerco quindi di capire quali sono le ridondanze con i metodi utilizzati fino ad ora.
 - E' possibile innanzitutto ridurre le dipendenze che presentano più attributi a destra a più dipendenze con un solo attributo a destra
 - E' possibile poi ridurre le dipendenze che hanno attributi estranei a sinistra a singole dipendenze. Sia $AX \rightarrow B$, A è un attributo estraneo se X^+ contiene B , ovvero se X da solo è sufficiente per determinare B
 - Si hanno adesso dipendenze irriducibili. L'ultimo passo è verificare se queste possono essere eliminate. Si ha che $X \rightarrow A$ è ridondante se tolta da F e poi calcolato X^+ , esso contiene A . E' ridondante quindi se X determina A anche senza la dipendenza funzionale appena rimossa.
- L'insieme F così costruito è quindi minimale, ovvero presenta un solo attributo a destra, sono assenti attributi estranei a sinistra e non possono essere tolte dipendenze senza perdere proprietà. Si definisce quindi copertura minimale di un insieme F un insieme equivalente a F ma di minore complessità.

L'algoritmo per il calcolo di M minimale per un insieme F è il seguente:

$$M = F$$

- ogni $X \rightarrow \{A_1, A_2, \dots, A_n\}$ è sostituita da $X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_n$
- ogni $X \rightarrow A$ è sostituita da $(X - \{B\}) \rightarrow A$ se $A \subseteq (X - \{B\})$ +
- ogni rimanente $X \rightarrow A$ in M è rimossa se $A \subseteq X$ anche in $\{F - \{X \rightarrow A\}\}$

$$F = \{A \rightarrow BC, B \rightarrow C, A \rightarrow B, AB \rightarrow C\}$$

$$F' = \{A \rightarrow B, A \rightarrow C, B \rightarrow C, \cancel{A \rightarrow B}, AB \rightarrow C\}$$

F'' = Va tolto perché c'è un percorso alternativo per arrivare a C da A e B

$$\hookrightarrow F'' = \{A \rightarrow B, A \rightarrow C, B \rightarrow C\}$$

$$F''' = \text{Tolgo } A \rightarrow C \text{ perché posso già arrivare a C da A}$$

$$\hookrightarrow F''' = \{A \rightarrow B, B \rightarrow C\}$$

La forma di Boyce-Codd

Le dipendenze funzionali talvolta possono presentare delle anomalie. Consideriamo ad esempio la dipendenza Impiegato \rightarrow Stipendio che rappresenta la seguente proprietà: "Lo stipendio di ciascun impiegato è funzione del solo impiegato, indipendentemente dai progetti a cui partecipa". Questa relazione in generale può presentare delle anomalie in quanto, non essendo l'attributo impiegato una chiave, possono verificarsi delle ripetizioni. La dipendenza funzionale Impiegato \rightarrow Progetto → Funzione non presenta invece questo tipo di problematiche in quanto Impiegato Progetto costituisce una chiave. Come abbiamo notato, le dipendenze funzionali che portano alla nascita di problemi sono quelle che al primo membro hanno attributi che non sono la chiave della relazione.

Si introduce quindi una prima forma normale, la forma normale di Boyce e Codd, in modo da non avere più questo tipo di anomalie. Si ha che una relazione R è in forma normale di Boyce-Codd se, per ogni dipendenza funzionale non banale $X \rightarrow Y$ definita su di essa, X è superchiave di R. Si ha quindi che, richiedendo che la parte sinistra sia una superchiave, vengono rappresentati soltanto concetti omogenei. I concetti indipendenti invece sono separati in altre relazioni, ed in questo modo non si generano più le anomalie viste in precedenza. Se un insieme F di dipendenze per R non è in BCNF, allora in F c'è almeno una dipendenza $X \rightarrow Y$ non banale con X superchiave di R.

Per verificare che una relazione sia in BCNF innanzitutto è utile tener presente il seguente teorema: Dato uno schema R ed un insieme F di dipendenze funzionali, se F non contiene alcuna $X \rightarrow Y$ non banale con X non superchiave di R, allora neanche F+ la contiene. Grazie a questo teorema è sufficiente analizzare le dipendenze non banali di F, o ancora meglio in una copertura minimale di F. Per far questo si utilizza un algoritmo di complessità polinomiale che verifica che ogni relazione a sinistra di ciascuna dipendenza funzionale sia una superchiave. L'algoritmo prima calcola la copertura minimale di F e poi verifica se gli insiemi di attributi a sinistra di ciascuna dipendenza sono superchiavi. In particolare, K è superchiave di R(Z) con dipendenze F se Z è contenuto in K+;

La qualità della decomposizione

Se una relazione non è in BCNF è possibile applicare il processo di decomposizione in modo da poterla rendere normalizzata. Il processo di decomposizione si basa sostanzialmente sul concetto di omogeneità dei dati già citato in precedenza. Relazioni che rappresentano concetti diversi devono essere quindi divise in relazioni più semplici che rappresentano soltanto concetti omogenei. Nella maggioranza dei casi pratici le decomposizioni avvengono come una per ogni concetto rappresentato dalla relazione, ma in alcuni casi può non essere necessario creare tante decomposizioni quante sono le dipendenze funzionali. Un esempio di decomposizione semplice può essere il seguente.

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

La decomposizione in generale può presentare 2 problemi principali: perdita di informazioni riscontrabile con il join e perdita delle dipendenze. Entrambe le problematiche devono essere risolte per poter parlare di una normalizzazione di qualità.

La perdita di informazioni si ha ogni qual volta non si riesca a ricostruire la tabella originaria a partire dalle sue proiezioni. La tabella originaria in generale deve essere costruita per mezzo di join a partire dalle sue decomposizioni e può capitare che in alcuni casi il join dia un risultato diverso da quello atteso, ovvero la tabella iniziale. Un possibile esempio di questo può essere un caso in cui la ricostruzione porta alla "nascita" di altre tuple non presenti nella relazione iniziale. In generale, data una relazione r su un insieme di attributi X, se X1 e X2 sono due sottoinsiemi di X la cui unione sia pari ad X stesso, allora il join delle due relazioni ottenute per proiezione r su X1 e X2, è una relazione che contiene tutte le tuple di r, più eventualmente altre tuple che vengono chiamate spurie. Si ha che r si decompone senza perdite se il join tra X1 e X2 da come risultato r stessa. La normalizzazione richiede che tutte le decomposizioni siano senza perdite. Di seguito un esempio di decomposizione con perdita

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

Impiegato	Sede
Rossi	Roma
Verdi	Milano
Neri	Milano

Progetto	Sede
Marte	Roma
Giove	Milano
Saturno	Milano
Venere	Milano

Impiegato	Sede
Rossi	Roma
Verdi	Milano
Neri	Milano

Progetto	Sede
Marte	Roma
Giove	Milano
Saturno	Milano
Venere	Milano

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

Si può formulare una condizione che garantisce la decomposizione senza perdita di una relazione. Sia r una relazione su X e siano X_1 e X_2 sottoinsiemi di X tali che la loro unione dia X ed X_0 l'intersezione di X_1 ed X_2 . Allora r si decomponete senza perdita su X_1 ed X_2 se soddisfa la dipendenza funzionale $X_0 \rightarrow X_1$ oppure $X_0 \rightarrow X_2$, ovvero se l'insieme degli attributi comuni alle due relazioni è chiave per almeno una delle relazioni decomposte. Nella decomposizione sopra indicata si ha che sede non è chiave di nessuna relazione, e per questo va perduta di informazioni. La condizione appena citata è solo sufficiente, in quanto è possibile che le informazioni vengano conservate seppure la condizione appena citata non sia verificata. Una cosa importante da notare di questa condizione sufficiente è che dal momento che essa è verificata riguarda tutte le istanze della relazione che prevede le dipendenze su cui essa vale, ed è quindi possibile dire che tutte le istanze di quella relazione si decompongono senza perdita.

→ Esiste un algoritmo che permette di decomporre una relazione in modo da renderla BCNF che è il seguente:

Decomponi(R,F):=

```
{ if esiste  $X \rightarrow A$  in F con X non superchiave di R
  then { sostituisci R con una relazione R1 con
        attributi U-A, ed una relazione R2 con
        attributi XUA;
  Decomponi(R1,FU-A);
  Decomponi(R2,FXUA)
  }
```

Per il funzionamento di questo algoritmo si ha che ogni dipendenza funzionale in F ha un unico attributo come membro destro e si ha che U è l'insieme di tutti gli attributi di R

L'algoritmo è corretto e per qualunque input esso riceva, termina e produce una relazione a partire da quella originaria che rispetta le seguenti caratteristiche: è in forma BCNF e non viene persa informazione nel join. Il problema di questo algoritmo sta nel creare le proiezioni delle relazioni iniziali su sottoinsiemi di attributi mantenendo la chiusura iniziale. Per far questo è infatti necessario un algoritmo di complessità esponenziale. Prima di esplicarlo è però necessario introdurre il concetto di proiezione di una dipendenza funzionale su un insieme di attributi. La proiezione di F su X, denotata da F_X , è l'insieme di dipendenze funzionali $Z \rightarrow Y$ in F che coinvolgono solo attributi in X, cioè tali che Z sia contenuto in X e Y sia contenuto in X. L'algoritmo che esegue la proiezione è il seguente:

CalcolaProiezione(F,X):=

```
{ result = Ø;
  per ogni sottoinsieme proprio S di X,
  per ogni attributo A in X tale che
    A non è in S, e
    non esiste alcun sottoinsieme S' di S
    tale che  $S' \rightarrow A$  è in result,
  if  $A \subseteq S$  then result = result U {  $S \rightarrow A$  };
  }
```

In alcuni casi la proiezione di F su X ha dimensione esponenziale rispetto alla dimensione di F ed X, come nel caso seguente:

- $R(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_n, C_1, C_2, \dots, C_n, D)$ e
- $F = \{ A_i \rightarrow C_i, B_i \rightarrow C_i \mid 1 \leq i \leq n \} \cup \{ C_1 C_2 \dots C_n \rightarrow D \}$
- La proiezione di F su {A₁, A₂, ..., A_n, B₁, B₂, ..., B_n, D} è
- $P = \{ X_1 X_2 \dots X_n \rightarrow D \mid X_i = A_i \text{ oppure } X_i = B_i \text{ per } 1 \leq i \leq n \}$

La complessità dell'algoritmo in generale è esponenziale in quanto non è possibile trovare insiemi equivalenti che abbiano minore complessità. Inoltre si ha che una variazione dell'ordine con cui si fa la decomposizione può far cambiare il risultato finale della decomposizione stessa.

→ Con le considerazioni di sopra abbiamo quindi risolto uno dei due problemi citati, ovvero quello della perdita di informazioni. Come avevamo già detto però, un altro problema che va eliminato in modo da ottenere delle decomposizioni di qualità è quello della perdita delle dipendenze. Questo accade quando le decomposizioni non hanno tutte le dipendenze che invece aveva la tabella originaria. Un modo per risolvere il problema è il seguente: per far sì che le decomposizioni presentino tutte le dipendenze funzionali che aveva la relazione iniziale, ciascuna dipendenza dello schema iniziale deve coinvolgere attributi che compaiono tutti insieme in uno degli schemi decomposti. Una decomposizione che soddisfa tale proprietà conserva le dipendenze.

Impiegato	Sede
Rossi	Roma
Verdi	Milano
Neri	Milano

Impiegato	Progetto
Rossi	Marte
Verdi	Giove
Verdi	Venere
Neri	Saturno
Neri	Venere
Neri	Marte

Questa decomposizione, effettuata tenendo di conto della DF Impiegato → Sede, mantiene le informazioni in quanto impiegato è chiave per la prima relazione, ma si vede invece che non mantiene le dipendenze. Sulla relazione iniziale era infatti presente la DF Progetto → Sede, che in questo caso non avrebbe permesso l'aggiunta dei campi "Neri" e "Milano", cosa resa invece possibile a causa dell'assenza della DF nelle due relazioni decomposte.

Come per la conservazione delle informazioni nel join, sono presenti alcuni metodi che permettono di conservare le dipendenze ed anche verificare se esse sono state conservative. Innanzitutto diciamo quando una decomposizione conserva le dipendenze. Sia R uno schema di una relazione con dipendenze funzionali F , e sia X un sottoinsieme di attributi di R . Si ha che la decomposizione di R in due relazioni con attributi X ed Y non provoca perdita di dipendenze se $F_X \cup F_Y$ è equivalente a $(F_X \cup F_Y) + = F +$, cioè è equivalente a F . La decomposizione con l'algoritmo citato precedentemente non assicura che non vi sia perdita di dipendenze come invece assicurava per la perdita di informazioni. Per verificare la condizione è necessario verificare se un insieme di dipendenze funzionali è equivalente ad un altro ed è necessario calcolare la proiezione di un insieme di dipendenze funzionali su un insieme di attributi che, come visto precedentemente, è molto dispendioso a causa della complessità esponenziale. Per la verifica di equivalenza è possibile utilizzare un metodo polinomiale che per ogni $X \rightarrow Y$ di F , calcola $X+$ rispetto a G e verifica se Y è dentro $X+$ e poi ripete lo stesso processo per $X \rightarrow Y$ in G e $X+$ rispetto ad F .

Caso importante: La relazione $R(A,B,C)$, con $F=\{A \rightarrow B, B \rightarrow C, A \rightarrow C\}$ non è in BCNF e la sua chiave è A . Innanzitutto è necessario determinare la sua copertura minimale che è data da $F=\{A \rightarrow B, B \rightarrow C\}$. A questo punto è possibile decomporla come $R1(A,B)$ ed $R2(B,C)$ partendo da $B \rightarrow C$, ottenendo in questo modo due relazioni in BCNF. In questo caso la decomposizione non perde nemmeno informazioni sul join e dipendenze funzionali.

Secondo quanto affermato fino ad ora, una decomposizione di qualità deve assicurare le seguenti caratteristiche:

- La decomposizione deve essere senza perdita di informazioni, in modo da poter ricostruire la relazione originaria. In questo caso le query sulle relazioni decomposte danno gli stessi risultati di quelle sulla relazione originaria.
- La conservazione delle dipendenze, in modo da non permettere aggiornamenti illeciti sulle relazioni decomposte. Le relazioni decomposte possono anche essere aggiornate separatamente da quella originaria, per i singoli concetti che esse rappresentano, a patto che rispettino le DF.

• La terza forma normale (3NF)

→ Generalmente, quando una relazione non può essere normalizzata in BCNF è presente un errore di progettazione. Esistono però alcuni casi, come il seguente, dove non è possibile raggiungere una BCNF di buona qualità secondo le caratteristiche elencate in precedenza.

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

Dirigente → Sede non rispetta la BCNF in quanto a sinistra non è presente una superchiave
La dipendenza Progetto Sede → Dirigente coinvolge tutti gli attributi e quindi non può essere conservata
→ La BCNF in questo caso non può essere raggiunta in nessun caso

→ Per casi come questo, dove la BCNF non può essere raggiunta, si utilizza una forma normale più lasca, chiamata terza forma normale. Una relazione r è in terza forma normale se, per ogni dipendenza funzionale non banale $X \rightarrow A$ definita su di essa, almeno una delle condizioni seguenti è verificata:

- X contiene una chiave K di r (condizione della BCNF)
- A appartiene ad almeno una chiave di r .

La relazione precedente soddisfa entrambe le caratteristiche, quindi non è in forma normale ma bensì è in terza forma normale. Come è possibile notare, sono presenti delle ripetizioni di "Dirigente", ridondanza che non sarebbe stata tollerata per la BCNF. La terza forma normale, essendo più lasca, può quindi avere problemi di presenza di alcune ridondanze. La terza forma normale presenta però alcuni vantaggi rispetto alla BCNF. In primis, qualsiasi relazione che non è in terza forma normale può essere decomposta in questo modo. In secondo luogo, verificare che una relazione sia in 3NF richiede un algoritmo deterministico NP-hard (esponenziale).

L'algoritmo in questo caso genera un sottoinsieme S di attributi di R che contiene A e controlla che S sia una chiave.

→ Vi sono 2 metodologie per decomporre in 3NF:

- 1° Modo: Viene calcolata la copertura minimale di F e poi viene applicata Decomponi(R, F) ottenendo gli schemi $R1(X1), R2(X2), \dots, Rn(Xn)$, ciascuno con le proprie dipendenze F_{Xi} . Adesso si considera l'insieme N delle dipendenze non preservate in $R1, R2, \dots, Rn$, cioè non incluse nella chiusura dell'unione dei vari F_{Xi} . A questo punto, per ogni dipendenza $X \rightarrow A$ in N si aggiunge lo schema relazionale $X A$ con le dipendenze funzionali relative a XA .
- 2° Modo: Innanzitutto è necessario dare una definizione più formale di 3NF. Uno schema di relazione $R(U)$ con l'insieme di dipendenze F è in 3NF se, per ogni dipendenza funzionale $X \rightarrow A \in F$, almeno una delle seguenti condizioni è verificata:

→ X contiene una chiave K di r, ovvero $Xf=U$

→ A è contenuto in almeno una chiave di r: esiste un insieme di attributi $K \subseteq U$ tale che $Kf=U$ e $(K-A)f \subset U$

A questo punto, l'algoritmo opera nel modo seguente:

- Si deriva la copertura minimale G di F.
- Si raggruppano le dipendenze in G in sottoinsiemi tali che ad ogni sottoinsieme Gi appartengono le dipendenze i cui membri sinistri hanno la stessa chiusura ($X \rightarrow A$ e $Y \rightarrow B$ appartengono a Gi se $X=Y$ secondo G)
- Si partizionano gli attributi U nei sottoinsiemi Ui individuati dai sottoinsiemi Gi del passo precedente. Se un sottoinsieme è contenuto in un altro si elimina.
- Si crea una relazione $R_i(U_i)$ per ciascun sottoinsieme U_i , con associate le dipendenze G_i .
- Si aggiunge una relazione per gli attributi che non sono coinvolti in alcuna FD
- Se non c'è già una relazione che contenga una chiave della relazione originaria, si aggiunge

Il secondo algoritmo è chiamato sintesi di schemi in 3NF ed è corretto perché: le relazioni sono in 3NF perché in ogni schema di relazione compaiono dipendenze che hanno primi membri equivalenti e quindi sono chiave, la conservazione delle dipendenze deriva dall'utilizzo di una copertura ridotta, grazie alla quale ogni dipendenza è necessaria e non può essere eliminata e la decomposizione senza perdita invece si ha dall'ultimo passaggio. A differenza del primo metodo che prima garantisce l'assenza di perdita sul join e conserva le dipendenze, il secondo metodo inverte questo ordine. Con entrambi questi metodi, si possono ottenere anche schemi in BCNF. Questo accade se la relazione ha una sola chiave.

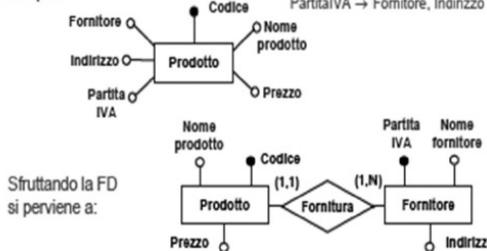
Le altre forme normali

- Oltre alle forme normali già citate esistono anche altre due tipologie di forme normali. Prima di dire quali sono è però necessario introdurre qualche altra definizione.
 - **Chiave candidata:** Se in uno schema di una relazione c'è più di una chiave, ognuna di esse è detta **chiave candidata** ed una di esse è la chiave primaria. Trovare le chiavi di una relazione richiede un algoritmo di complessità esponenziale.
 - **Attributo primo:** Un attributo di R è detto **attributo primo** di R se è membro di una qualche chiave candidata di R. Il problema di stabilire se un attributo è primo è NP-completo.
 - **DF completa:** Una DF $X \rightarrow Y$ è completa se la rimozione di un qualsiasi attributo A da X comporta che la DF non valga più.
 - **DF parziale:** E' possibile rimuovere attributi da X e la DF continua a rimanere valida
- Una volta date queste definizioni è possibile dire quali sono le altre forme normali e dare nuove definizioni per quelle già conosciute:
- **Prima forma normale:** Si basa sostanzialmente sulla definizione di modello relazionale e richiede che il dominio di un attributo comprenda soltanto valori atomici e che il valore di qualsiasi attributo in una tupla sia un valore singolo del dominio
 - **Seconda forma normale:** Una relazione R è in seconda forma normale se ogni attributo non primo A di R ha una DF completa con ogni chiave di R. Possono esistere dipendenze con attributi non primi.
 - **Terza forma normale:** Una relazione R è in terza forma normale se, per ogni FD non banale $X \rightarrow Y$ definita su R, si ha che X è superchiave di R od ogni attributo in Y è primo.
 - Si ha che la BCNF implica sia la 3NF che la 2NF, implicata sempre dalla 3NF.

Normalizzazione e ristrutturazioni

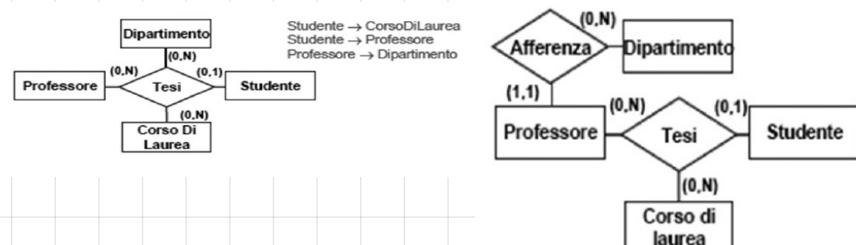
- La normalizzazione, pur non essendo una tecnica di progetto, permette di verificare la qualità del progetto in corso già a partire dalla fase di progettazione concettuale e logica. Nella fase di progettazione logica, applicando la normalizzazione, è infatti possibile verificare eventuali errori commessi nella precedente progettazione logica. Partiamo dall'utilizzo della normalizzazione nella progettazione concettuale.

Esempio:



Nella progettazione concettuale è possibile considerare le entità e le associazioni come tabelle. Le entità sono relazioni che hanno come attributi quelli dell'entità e la verifica della normalizzazione può essere fatta prendendo in considerazione le dipendenze funzionali che riguardano gli attributi dell'entità, e che in generale devono avere a sinistra l'identificatore della stessa.

La DF $\text{Partita IVA} \rightarrow \text{Fornitore Indirizzo}$ ha a sinistra un elemento che non è l'identificatore, mentre a destra attributi che non fanno parte della chiave. L'entità quindi va decomposta in quanto si capisce che rappresenta due concetti diversi. La decomposizione ottenuta rispetta le due proprietà di qualità delle decomposizioni.



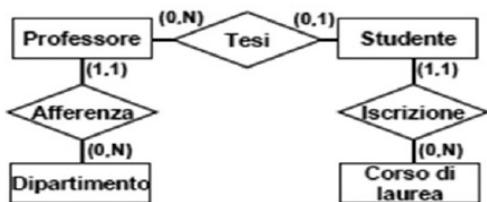
Una volta analizzate le entità vanno analizzate le associazioni narie, che spesso non rispettano la normalizzazione in 3NF.

In questo caso la DF che non rispetta la 3NF è $\text{Professore} \rightarrow \text{Dipartimento}$, in quanto è indipendente dalla quantità di studenti che svolgono la tesi. L'associazione unisce quindi concetti disomogenei e va divisa.

Studente → CorsoDiLaurea (iscrizione)

Studente → Professore (per chi ha un relatore)

- È quindi opportuno procedere a un'ulteriore ristrutturazione:



L'associazione inoltre può essere di nuovo divisa. Infatti si ha che le proprietà descritte dalle dipendenze Studente→Professore e Studente→CorsoDiLaurea sono indipendenti perché non tutti gli studenti hanno un relatore. In generale, è conveniente dividere associazioni non binarie che a destra hanno più di un termine. In particolare, è opportuno dividere quelle che a sinistra hanno un termine di un'entità e a destra quelli delle altre 2. Non è conveniente se le dipendenze non possono essere mantenute nella decomposizione o se le dipendenze sono correlate tra loro.

Le transazioni

• Introduzione alle transazioni

→ La gestione delle transazioni è fondamentale per l'implementazione di una base di dati. Gestendo le transazioni, in modo che la base di dati sia **consistente e persistente**, il DBMS garantisce l'affidabilità e **gestisce la concorrenza** tra le transazioni, in quanto il modello può avere **molti utilizzatori ed un solo server**, come avviene nella maggioranza dei casi. Un database può infatti avere una serie di tabelle per gestire una grande varietà di problematiche relative ad un'azienda. Nella gestione di tutte queste tabelle, il primo problema è la **quantità degli utenti** che si occupano della gestione. Nel **sistema informativo aziendale**, un database può avere **molti utenti**, quindi il gestore della base di dati deve considerare questo aspetto.

→ Un computer gestisce il fatto di avere più **richieste contemporaneamente** con il **multitasking**, che consente di eseguire **più operazioni nello stesso momento**. Esso infatti dedica una **certa quantità di tempo di tempo a ciascuna richiesta**. Il sistema operativo dedica quindi uno **slot a ciascun utente** e poi lo revoca una volta terminata la richiesta. Se vi sono **più CPU** l'esecuzione può essere **contemporanea**, mentre nel caso di una **singola CPU** l'esecuzione è **interleaved**.

→ In questo assetto del sistema la **transazione** è un'unità di lavoro elementare svolta dalle **applicazioni** che **non deve essere interrotta** e che quindi deve essere considerata come **un tutt'uno**. Il computer sa eseguire le operazioni di basso livello e traduce le operazioni di alto livello in modo da renderle eseguibili.

→ Le transazioni sono un insieme di operazioni elementari raggruppate in un'unità logica caratterizzate da una serie di operazioni sulla base di dati. Ognuna di queste corrisponde ad una lettura o scrittura sulla **base di dati**. I sistemi di **gestione sono transazionali**, perché permettono di **gestire le transazioni**, che possono comprendere l'utilizzo di più **parti della base di dati** e possono anche **gestire la concorrenza** data dall'**utilizzo contemporaneo** da parte di **più utenti in contemporanea**.

• Le transazioni in SQL

→ Dal punto di vista del linguaggio, la transazione è una **parte di codice SQL** racchiusa tra **due delimitatori** che simboleggiano l'**inizio e la fine della transazione**. La transazione inizia con **start transaction** e contiene varie istruzioni che però vengono raggruppate in un'unica unità logica. La fine non ha un solo comando specifico, ma bensì esistono **due comandi**: il comando **commit work** che richiede che la transazione si concluda con **esito positivo** e che quindi gli **effetti della transazione vengano salvati** all'interno della base di dati, oppure il comando **rollback work**, che **annulla di fatto tutti gli aggiornamenti** provocati dalla transazione. L'istruzione **rollback work** viene spesso utilizzata in concomitanza ad **istruzioni di controllo**. L'applicazione definita può essere un insieme di transazioni, quindi un utente può **eseguire più transazioni**. Una transazione che soddisfa la struttura precedente si dice **ben formata**.

• Le proprietà ACID delle transazioni

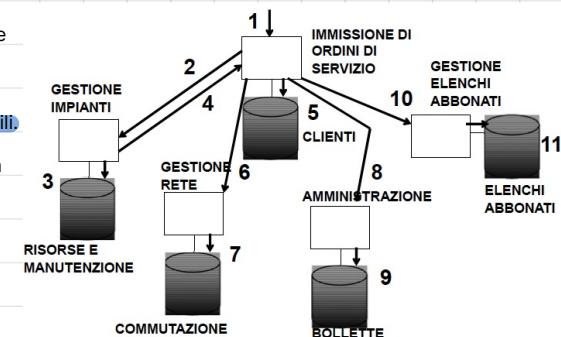
→ Le transazioni devono garantire tutte le proprietà **ACID**, ovvero **atomicità, consistenza, isolamento e persistenza**.

→ L'**atomicità** garantisce che le operazioni all'interno della singola transazione di fatto sono **viste come un'unica unità logica indivisibile** quindi durante l'esecuzione o tutto ha successo o tutto fallisce. La transazione non rende visibili le operazioni **intermedie** che essa esegue sui dati, quindi prima della terminazione con successo (commit) non si effettuano modifiche sulla **base di dati**. In caso di errori quindi la transazione non lascia mai la base di dati in uno stato intermedio ed il gestore ricostruisce la **situazione iniziale della base di dati (undo)**. Dopo il commit invece i dati diventano persistenti, quindi il gestore della base di dati deve considerare anche la possibilità di dover rifare le operazioni della transazione (redo). Per quanto riguarda il comando **rollback work**, di fatto si ha un "suicidio" deciso autonomamente dalla transazione, mentre possono capitare casi in cui vengono riscontrati errori per i quali la transazione non può essere portata a termine e per questo il sistema la "uccide".

→ La **consistenza** garantisce che **ogni operazione** deve lasciare la base di dati in **stato consistente**. Il problema della consistenza è quello che, per ogni transazione, i **vincoli di integrità** devono essere rispettati. Durante le transazioni possono essere commessi degli errori in questo senso, ma non possono essere riportati nel risultato finale. I **vincoli di integrità** di tipo **immediato** vengono verificati direttamente all'interno dell'esecuzione della transazione e vengono eliminati gli errori senza necessariamente portare ad un **abort**, mentre i **vincoli di tipo diffuso** vengono verificati alla fine della transazione in extremis e portano all'abort della transazione.

→ L'**isolamento** permette al **singolo utente** di vedere il **database** come se fosse l'unico utilizzatore, quindi **indipendentemente da altri utenti**. Dovendo garantire l'isolamento, ogni transazione non può influenzarne altre, in quanto ogni transazione vede uno **stato iniziale dei dati** e deve darne uno **finale**, indipendentemente dalle transazioni effettuate da altri utenti. Ogni transazione quindi non può comportare il **rollback** di altre come un effetto domino.

→ La **durabilità** invece prevede la **persistenza**, ovvero che se una **transazione termina con un commit**, i suoi **effetti vengono memorizzati** all'interno della base di dati. In ogni caso, i dati non vanno persi in nessun caso., in quanto dopo ogni guasto i dati possono sempre essere recuperati.



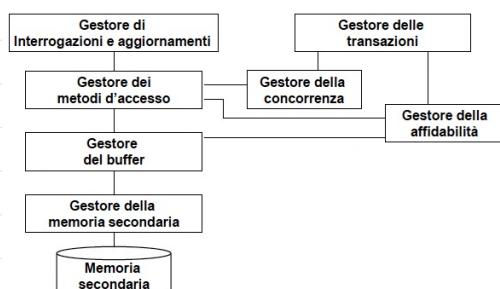
```
start transaction;
update ContoCorrente
    set Saldo = Saldo + 10 where
        NumConto = 12202;
update ContoCorrente
    set Saldo = Saldo - 10 where
        NumConto = 42177;
commit work;
```

Una transazione è sempre in uno stato preciso. Una transazione è in **state active** dopo il **begin transaction** ed è lo stato in cui è possibile effettuare **operazioni di lettura e scrittura**. Una transazione è invece **partially committed** se è stata eseguita l'istruzione che ne determina la fine, però il gestore deve ancora determinare il fatto che non vi siano errori. Se non ve ne sono, la transazione diventa **committed**. La transazione è invece in stato **failed** se l'esecuzione non può essere conclusa in modo normale, mentre è **aborted** se ha subito un **rollback**, ovvero è stato ripristinato lo stato precedente alla transazione.

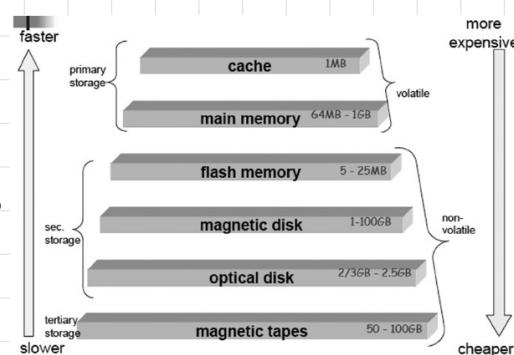
I database NoSQL non garantiscono l'affidabilità garantita invece dai database relazionali, quindi non possono essere utilizzati in alcuni ambiti, come per esempio nei database di una banca, in quanto ogni transazione in questo caso deve garantire tutte le proprietà ACID.

La memoria principale e secondaria

Il DBMS è composto da vari moduli, tra cui il gestore di affidabilità, che si occupa di **garantire affidabilità e consistenza**, il gestore della concorrenza, che permette di gestire più operazioni contemporaneamente ed il gestore di integrità. In generale, si ha un gestore delle transazioni, che traduce le istruzioni e le ottimizza in modo da renderle più efficienti. Esse vengono eseguite a livello più basso dal gestore dei metodi di accesso, che si interfaccia con la memoria di archiviazione del database attraverso il gestore del buffer. Questo gestisce lo scambio di dati tra **memoria principale e secondaria** e invia le richieste di lettura e scrittura verso quest'ultima tramite il gestore della memoria secondaria. Il database, in quanto a memoria presenta due tipologie di memorie. La memoria principale è **volatile** ed è molto **veloce**, mentre la memoria secondaria non lo è, in quanto i dati vengono memorizzati in modo **persistente**.



Il database richiede però memoria **secondaria** date le dimensioni del database e in quanto la memoria principale è volatile ed è però necessario memorizzare i dati in modo persistente. I dati vengono organizzati in **blocchi di lunghezza fissa** (poche decine di KB) e le operazioni che possono essere eseguite sono quelle di lettura e scrittura di un blocco. Il tempo di lettura e scrittura in memoria secondaria è molto maggiore di quello necessario per l'accesso alla memoria principale, che quindi può essere trascurato. Il tempo di accesso alla memoria secondaria è dato da 3 fattori: **posizionamento della testina**, **tempo di trasferimento** e **tempo di latenza**. Il primo riguarda il posizionamento della testina sulla parte del disco dove è presente il dato, mentre l'ultimo riguarda l'attesa necessaria perché il blocco ricercato ruoti sotto la testina. È quindi facile capire che gli accessi sono molto più veloci se i blocchi sono allocati in posizioni contigue.



La memoria principale è invece organizzata in pagine.

Il buffer

Non è possibile trasferire tutta la base di dati nella memoria principale, e quindi si trasferiscono soltanto alcuni dati utili in quel momento. Il buffer permette quindi di trasferire questi dati, ed è costituito da un'area di memoria gestita dal DBMS e condivisa tra le applicazioni che effettuano le transazioni. In quanto ad organizzazione, come la memoria principale, anche il buffer è suddiviso in pagine di dimensioni uguali o multiple ai blocchi di byte della memoria secondaria. Una volta trasferiti i dati nel buffer, le operazioni vengono fatte su questa memoria. Il gestore del buffer accetta le richieste di operazioni sui dati e quando questi non sono presenti se li fa inviare dalla memoria secondaria. La sua funzione è quindi quella di restituire una locazione di una pagina su cui lavorare. I dati presenti all'interno del buffer non comprendono soltanto i dati utili alla singola operazione, ma anche alcuni dati ausiliari. In ogni caso, viene utilizzato il principio di **località dei dati**, che afferma che i dati utilizzati di recente hanno maggiore probabilità di essere referenziati anche in futuro. Un'altra regola che viene utilizzata è quella che afferma che il 20% dei dati è acceduto dall'80% delle applicazioni. Per questo è presente un direttorio che contiene il contenuto del buffer, con file fisico e numero di blocco corrispondente. Inoltre, prevede variabili di stato che forniscono informazioni sull'utilizzo corrente della pagina o sullo stato della sua modifica.

Per richiedere memoria al buffer si utilizzano primitive. Le operazioni infatti richiedono una pagina tramite la primitiva **fix**, che risponde con un **indirizzo del buffer** e, nel caso in cui i dati non siano presenti, si occupa di trasferirli. **Unfix** invece dichiara che la transazione è finita e che i dati non servono più, azzerando il contatore di utilizzo della pagina. La primitiva **setDirty** indica al gestore che la pagina è stata modificata, modificando di conseguenza anche il contatore corrispondente. La primitiva **force** permette di trasferire i dati nella memoria secondaria in modo sincrono se la transazione ha fatto **commit**. Alternativamente è possibile utilizzare **flush**, che dà al gestore la libertà di scegliere il momento in cui memorizzare le informazioni in memoria secondaria.

Quando viene richiesta una certa pagina, la **fix** verifica se la pagina è presente nel buffer. Se c'è, il contatore della pagina viene incrementato e viene restituito l'**indirizzo della pagina**. Nel caso che non sia presente, è necessario individuare una posizione dove poter creare una nuova pagina e trasferire i dati. Una volta trovata, i dati vengono caricati dalla memoria secondaria alla nuova pagina e ne viene restituito l'indirizzo. Se non c'è memoria, deve essere eliminata una pagina per creare spazio. Per questo viene utilizzata una politica di rimpiazzamento. Se la pagina da eliminare è stata modificata, va forzato il contenuto nel database tramite la **force**. Dopo aver fatto questo, vengono posti i nuovi contenuti nella pagina detta **vittima**. La politica più utilizzata è la **LRU**, ovvero **Less Recently Used**. Viene quindi eliminata la pagina usata meno recentemente, con una politica che quindi è **FIFO**. Se non vi sono pagine che non sono in uso, la transazione fallisce. La strategia **steal** prevede che si possa scegliere una pagina che è stata modificata precedentemente. In questo caso viene fatta una **flush** e vengono inseriti i nuovi dati. Per quanto riguarda la tecnica **no-steal** invece se non è presente nessuna pagina che non è mai stata modificata, la transazione viene posta in **una coda di esecuzione**. In ogni caso, la pagina correntemente analizzata viene contrassegnata come in uso.

→ Vi sono altre strategie per la gestione dei buffer: la strategia **force** prevede che i dati vengano portati in memoria secondaria ad ogni commit, mentre la strategia **no-force** lascia questa libertà al DBMS che li trasferisce in memoria in un momento di maggiore convenienza. Il gestore per esempio li trasferirà in memoria in un momento in cui vi sono meno transazioni.

→ I DBMS in generale utilizzano il **file-system per le operazioni**, ma al contempo creano una propria astrazione dei file in modo da garantire maggiore efficienza. In generale però, le **funzionalità del sistema operativo** utilizzate sono poche e sono limitate ad una serie **limitata di operazioni**. Per far questo il DBMS **crea dei file grandi** che possono essere visti come una sorta di spazi di memoria alternativi in cui il DBMS può costruire la sua struttura interna.

• I malfunzionamenti e la memoria stabile

→ Vi sono vari tipi di malfunzionamenti a cui un database può andare incontro. I

malfunzionamenti della memoria secondaria riguardano i dischi, mentre altri

malfunzionamenti riguardano la parte di memoria volatile, come per esempio la caduta di corrente. Vi sono infine gli **errori logici**, che porterebbero la base di dati in stato

inconsistente, dati per esempio da errori software. I problemi della memoria **non volatile**

riguardano la memorizzazione dei dati, che possono essere persi a causa di

malfunzionamenti a seguito di crash dei dischi o dei nastri. I dati contenuti nella memoria

volatile possono essere persi durante una transazione a causa di alcuni eventi, quali

l'interruzione di corrente. Vi è poi la **memoria stabile**, che è una parte di memoria che non

può essere persa. La memoria stabile è necessaria per il **funzionamento del controllore**

dell'affidabilità. Essa è **un'astrazione**. Non esistono infatti memorie che abbiano possibilità

nulla di rompersi, tuttavia vengono utilizzati **alcuni metodi per minimizzare questa possibilità**.

Uno dei metodi ritenuto affidabile è l'utilizzo **di nastri**, mentre un altro metodo ritenuto valido è l'utilizzo di **due unita disco in modalità mirrored**, ovvero l'utilizzo di due dischi con le stesse informazioni memorizzate al di sopra. La strategia utilizzata è quindi quella della **ridondanza**.

• Il log

→ L'elemento fondamentale utilizzato dal gestore dell'affidabilità è il log. Il **log** è un file **replicato** che riporta tutte le

operazioni fatte nell'ordine in cui sono state fatte. È quindi un **file sequenziale**, nel quale gli ultimi record inseriti

caratterizzano il **blocco corrente**, che può essere sostituito da record inseriti **successivamente**. I record inseriti nel

log sono di 2 tipi: **di transazione e di sistema**. I record di **transazione** prevedono **start, commit ed abort** e anche

altri log corrispondenti alle operazioni effettuate all'interno della transazione. I record di **sistema** comprendono

invece i **checkpoint e i dump**. La struttura dei log di **begin, commit e abort** prevede l'**indicazione dell'operazione**

effettuata ed il **nome della transazione**, quindi si ha **B(T), C(T), A(T)**. I record di **update** invece contengono anche

l'**oggetto su cui è fatto l'update** e l'**indicazione di before state ed after state**, ovvero il valore dell'oggetto prima e

dopo l'aggiornamento. Si avranno quindi record del tipo **U(T,O,BS,AS)**. I record di **insert e delete** invece

presentano solo l'**indicazione di after o before state**, e quindi saranno del tipo **I(T,O,AS)** e **D(T,O,BS)**. Questi record

permettono di **disfare o rifare azioni** sulla base di dati mediante **undo e redo**. Per effettuare una **undo su un**

oggetto basta ricopiare in esso il valore che aveva in **BS**, mentre basta **eliminarlo nel caso di una insert**. Per

effettuare una **redo** invece basta ricopiare nell'oggetto il valore di **AS**, mentre per **rifare una delete** basta **eliminare**

l'oggetto. Si ha che **undo(undo(A))=undo(A)** e **redo(redo(A))=redo(A)**.

→ La struttura del log permette di recuperare lo stato consistente in ogni caso in quanto esso mantiene in memoria

tutto il funzionamento del database. I **checkpoint ed i dump** permettono di sezionarlo in modo da **non doverlo**

ripercorrere sempre dall'inizio. A seconda del guasto è quindi sufficiente **ripartire dal checkpoint o dal dump più**

vicino. I **checkpoint** servono per capire da dove partire per recuperare la **base di dati in caso di guasti**. Nel

checkpoint vengono poste le **transazioni a metà**, in quanto le transazioni **non terminate** non potrebbero essere

recuperate perché non avrebbero ancora fatto il **commit**. In questo modo è possibile recuperare le transazioni da

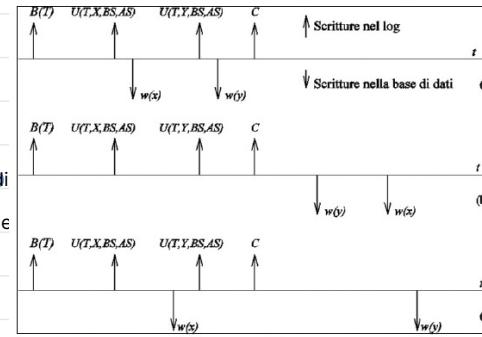
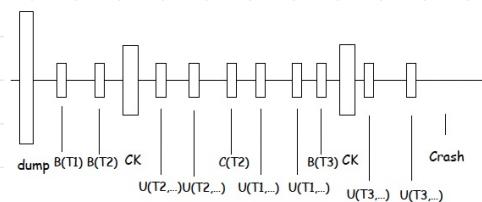
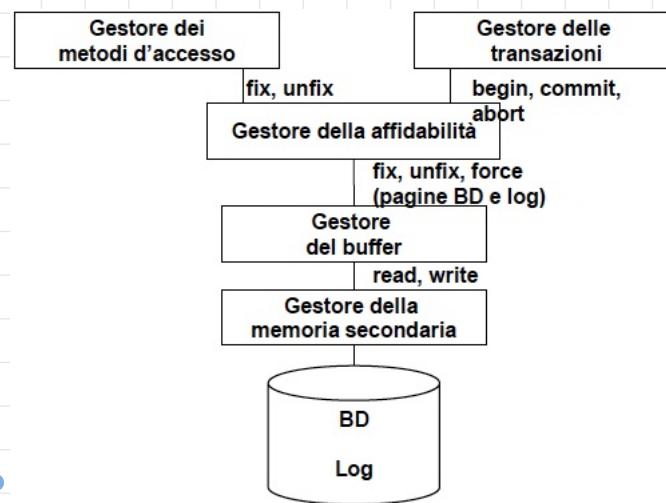
eventuali crash. La procedura per generare un **checkpoint** è la seguente: il gestore **smette di accettare richieste di**

scrittura, abort e commit, in quanto si altererebbe la veridicità del **checkpoint**. Vengono poi **trasferite in memoria le**

pagine di buffer già modificate da transazioni che hanno fatto il commit e viene scritta nel log con una force una

lista delle transazioni attive. Una volta fatto questo, il sistema riprende a funzionare.

→ Il **dump** invece è una **copia della base di dati** nella sua **interezza**, che viene fatta quando il sistema **non è attivo**, in quanto è eseguita in **mutua esclusione** con tutte le altre operazioni, anche se i sistemi più moderni prevedono tecniche di **hot backup** che non richiedono lo stop del funzionamento. Le **copie** vengono realizzate **su vari server**.



→ Per garantire la correttezza del log vengono utilizzate due regole. La prima regola è la Write-Ahead-Log, che prevede la scrittura dell'operazione nel log prima dell'effettuazione dell'operazione stessa. In questo modo è possibile effettuare undo in quanto è disponibile il valore prima dell'operazione. La seconda regola utilizzata è la regola di commit-precedenza che prevede la scrittura dell'after state prima del commit, permettendo di completare operazioni che hanno effettuato commit ma i cui dati non sono stati ancora salvati in memoria. In pratica le regole sono più semplici perché i record vengono scritti nel log e poi vengono fatte le operazioni sui dati, memorizzando sia il before che l'after state. La memorizzazione dei record di commit viene fatta in modo sincrono, mentre i record di abort vengono memorizzate in modo asincrono. Una transazione attiva viene posta nel checkpoint e termina completamente quando il record di commit viene posto nel log. Se c'è un guasto prima di inserire il commit, la transazione viene disfatta completamente. Se il guasto è subito dopo il commit nel log la transazione viene considerata corretta e vengono fatte tutte le operazioni per recuperare i dati ed inserirli all'interno del database.

→ Le operazioni fatte da una transazione possono essere riportate nella base di dati in vari modi. Con la modalità immediata i dati vengono riportati in memoria durante l'esecuzione della transazione. Quando una transazione incorre in un guasto ed è ancora attiva, l'undo prevede che venga cancellato l'after state e venga ripristinato il before state. In questo caso non è previsto il redo. La modalità differita prevede che i dati vengano memorizzati solo dopo il commit, rendendo quindi non necessario un undo. Essa richiede redo, ma l'abort non richiede nessuna operazione. La modalità mista prevede una combinazione delle due modalità precedenti, e quindi richiede sia operazioni di undo che di redo. La modalità differita non è molto utilizzata perché avere una procedura di recovery più leggera non è significativo se questa blocca temporaneamente le operazioni di lettura e scrittura dei dati. Questo accade perché i casi di crash sono meno frequenti. E' quindi più conveniente avere una procedura di gestione ordinaria più efficiente di quella di recovery. Per semplificare la gestione ordinaria il gestore decide quando fare le scritture nella base di dati, mantenendo però che le scritture delle operazioni e del commit vengono fatte prima nel log e poi nella base di dati, rendendo sempre possibili le operazioni di undo e redo.

→ Per il funzionamento normale delle transazioni è necessaria anche l'operazione di rollback che permette l'abort di una transazione. Quando una transazione va in stato abort, tutte le sue operazioni devono essere cancellate, di fatto come se la transazione non fosse mai stata fatta, in quanto il gestore potrebbe già aver scritto qualcosa nella base di dati. Nel caso di rollback, possono verificarsi 2 casi: nel primo caso l'abort scatena immediatamente l'undo e quindi le operazioni devono essere disfatte subito e devono essere mantenute nel log in quanto in caso di crash devono essere ripetute, mentre nel secondo caso l'abort viene posticipato in fondo e viene inserito il record di abort nel log.

→ La gestione degli undo è influenzata anche dalle scelte fatte per il rimpiazzamento. Se la politica scelta è no-steal, ovvero non sono rimpiazzati dei dati che sono stati modificati non è necessario fare nulla. Se invece viene usata la force, le transazioni che hanno fatto commit hanno già riportato i dati della base, quindi non è necessario fare redo di queste transazioni. Lo svantaggio in questo caso è la necessità di molti trasferimenti. La strategia utilizzata normalmente è la no-force combinata con la strategia steal, in quanto garantisce la maggiore efficienza lasciando la maggiore libertà di decisione al gestore del buffer.

• La ripresa a caldo e a freddo

→ Per quanto riguarda i guasti che una base di dati può riscontrare, ve ne sono due tipi. Vi sono guasti in cui si perde la memoria volatile, detti anche guasti soft o guasti di sistema, e guasti in cui si perde la memoria non volatile, detti guasti hard o guasti di dispositivo. Nel primo caso di guasti il gestore effettua una ripresa a caldo, mentre nel secondo caso il sistema fa una ripresa a freddo. Il metodo utilizzato per rimediare ai guasti è il metodo fail-stop. Inizialmente vengono arrestate tutte le transazioni ed il sistema operativo viene riavviato ripristinandone la completa funzionalità. A questo punto viene avviata la procedura di restart ed al termine il buffer è vuoto, però le transazioni sono in grado di ripartire dal punto in cui erano rimaste.

→ Il processo di restart inizia con una classificazione delle transazioni. Vi sono delle transazioni complete, transazioni attive che hanno raggiunto il commit, e per queste è necessario rifare le azioni al fine di garantire la persistenza e transazioni che non hanno raggiunto il commit, per le quali è necessario annullare le azioni effettuate.

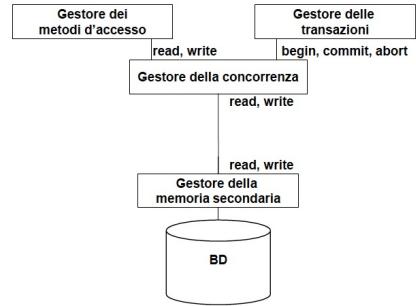
→ Per la ripresa a caldo si accede all'ultimo blocco noto prima del guasto e poi si risale al checkpoint che contiene le transazioni attive in quel momento. Le transazioni attive a quel momento sono quelle che non avevano fatto commit. Vengono creati un insieme di UNDO ed un insieme REDO. L'insieme UNDO viene inizializzato con le transazioni attive al checkpoint e l'insieme REDO è vuoto. Una volta fatto questo si percorre il log in avanti e vengono aggiunti ad UNDO le transazioni che hanno un record di begin, e vengono scambiate da UNDO a REDO le transazioni che hanno un record di commit. Vengono adesso disfatte tutte le transazioni presenti in UNDO, risalendo fino alla transazione più vecchia nei due insiemi. Infine vengono fatte le azioni delle transazioni in REDO in ordine di inserimento. In questo modo viene garantita sia l'atomicità, in quanto le operazioni incerte non vengono memorizzate, e la persistenza, in quanto le transazioni che fanno commit vengono effettivamente trasferite in memoria di massa.

→ La ripresa a freddo avviene invece nel modo seguente. Se vengono persi dati sulla memoria secondaria viene richiesto il dump. In questo caso si risale al dump più vicino, che contiene la copia di riserva. Vengono fatte quindi le operazioni sugli oggetti in modo da recuperare il loro stato al momento del guasto. Vanno quindi ripetute tutte le operazioni fatte sugli oggetti a partire dal dump fino al momento del guasto. Le azioni eseguite sono quelle riguardanti la base di dati, le azioni di commit e le operazioni di abort. A questo punto viene fatta la ripresa a caldo, dopo aver ripristinato gli oggetti che avevano perso uno stato consistente. In questo modo la base di dati è riportata in uno stato consistente.

La concorrenza

Introduzione al gestore e problematiche comuni

- Le transazioni sono i caratterizzanti principali del carico di un database. Il carico applicativo di un database viene infatti misurato in transazioni per secondo o tps. Essendo che i database devono gestire un gran numero di tps, non è possibile gestire tutte in serie e si rende quindi necessaria la concorrenza. Per gestire la concorrenza è quindi necessario un ulteriore gestore, oltre a quelli già noti, detto gestore della concorrenza. Il gestore della concorrenza riceve le richieste di accesso ai dati sia in lettura che in scrittura e decide se accettarle o meno, eventualmente riordinandole. Essendo che esso decide il loro ordine, il gestore della concorrenza viene anche detto scheduler. Si indicano con $r(x)$ e $w(x)$ le operazioni di lettura e scrittura della pagina dove è inserito il valore x .
- Esistono 5 tipologie di anomalie diverse che possono verificarsi. Il primo caso è la perdita di aggiornamento. Per spiegare questa anomalia consideriamo due transazioni t_1 e t_2 identiche: $t_1: r(x), x = x + 1, w(x)$ e $t_2: r(x), x = x + 1, w(x)$. Nel caso in esempio, se si eseguono le transazioni in sequenza, alla fine x avrà come valore 4, mentre se le transazioni sono concorrenti, x assume come valore finale 3 in quanto gli effetti della transazione t_2 vengono persi.
- La seconda anomalia presa in esame è la lettura sporca. In questo caso il valore di x al termine dell'esecuzione è 2, in quanto la transazione t_1 ha effettuato abort, e quindi tutte le modifiche che essa ha fatto sui dati sono state annullate, mentre la transazione t_2 legge 3 in quanto il valore di x era stato modificato dalla transazione t_1 prima di fare abort. Il termine lettura sporca deriva dal fatto che la transazione t_2 legge uno stato intermedio di x che invece non avrebbe dovuto leggere. Per rendere corretta la transazione sarebbe stato necessario annullare tutte le transazioni che avessero letto dati modificati da t_1 , creando però un effetto domino di difficile soluzione.
- La terza anomalia riguarda le letture inconsistenti. Nell'esempio di questo caso si ha che la transazione t_1 legge due valori di x , uno prima ed uno dopo il commit di t_2 , che sono diversi. Questo non deve però accadere.
- Il quarto caso preso in esame è l'aggiornamento fantasma. Nell'esempio considerato si ha che la transazione t_2 non viola i vincoli di integrità, mentre al momento del suo commit nella variabile s della transazione t_1 si ha il valore 1100 che viola invece questi vincoli.
- L'ultimo caso è l'inserimento fantasma. Supponiamo che la transazione t_1 si occupi di leggere gli stipendi degli impiegati e calcolarne la media. Se viene inserito un nuovo impiegato da una transazione t_2 , il valore della media degli stipendi cambia, portando ad un'anomalia nella transazione t_1 .



Lo schema delle

- Per quanto riguarda la gestione della concorrenza, si hanno varie metodologie di gestione. La politica **read uncommitted** evita la perdita di aggiornamento, mentre la **read committed** evita anche la lettura sporca. Le ultime due politiche sono la **repeatable read**, che evita tutte le anomalie eccetto l'inserimento fantasma e la **serializable** che evita tutte le anomalie già descritte.
- Una transazione è un oggetto sintattico del tipo $t_1: r_1(x)r_1(y)w_1(x)w_1(y)$ e vengono eseguite tramite uno **schedule**, ovvero una sequenza di operazioni di lettura e scrittura richieste in modo concorrente da più transazioni. Un esempio di schedule può essere quindi $S_1: r_1(x)r_2(z)w_1(x)w_2(z)\dots$. Non si ha quindi un'esecuzione sequenziale delle transazioni, ma si ha invece un'esecuzione interleaved nella quale le azioni vengono eseguite nell'ordine in cui esse compaiono all'interno dello schedule. Il controllo della concorrenza viene fatto dallo **scheduler** che decide se le azioni richieste da una transazione debbano essere eseguite, rifiutate, oppure semplicemente essere messe in attesa. Per semplicità, si effettua una **commit-proiezione** dello schedule considerando soltanto le operazioni relative a transazioni che effettuano **commit**. In ogni caso, il risultato finale dello schedule deve però essere analogo ad un'esecuzione seriale delle transazioni.

Schedule seriali e view - serializzabili

- Uno schedule si definisce **seriale** se le operazioni di ciascuna transazione sono eseguite senza essere inframmezzate da operazioni di altre transazioni. Un esempio di schedule seriale è il seguente: $S_2: r_0(x)r_0(y)w_0(x)r_1(y)r_1(x)w_1(y)r_2(x)r_2(y)r_2(z)w_2(z)$. L'esecuzione di uno schedule S_i è corretta se porta allo stesso risultato di uno schedule seriale S_j definito dalle stesse transazioni di S_i . Nel caso in cui questo accade lo schedule S_i viene detto **serializzabile**. Perché questo funzioni è necessario introdurre il concetto di **equivalenza** tra schedule. Per introdurre questo concetto introduciamo altri due concetti importanti, la relazione "legge da" e la nozione di scrittura finale. Si dice che $r_i(x)$ legge da $w_j(x)$ quando $w_j(x)$ precede $r_i(x)$ e non ci sono $w_k(x)$ tra $r_i(x)$ e $w_j(x)$. Si dice poi che $w_i(x)$ è una scrittura finale se nello schedule non compaiono altre operazioni di scrittura che riguardano x . Due schedule sono quindi **view-equivalenti** se possiedono la stessa relazione "legge da" e le stesse scritture finali. Da questo possiamo dire che uno schedule si dice **view-serializzabile** se esiste uno schedule seriale **view-equivalente** ad esso. Un esempio è il seguente:

$S_3: w_0(x)r_2(x)r_1(x)w_2(x)w_2(z)$

$S_4: w_0(x)r_1(x)r_2(x)w_2(x)w_2(z)$

$S_5: w_0(x)r_2(x)w_2(x)r_1(x)w_2(z)$

$S_6: w_0(x)r_2(x)w_2(x)w_2(z)r_1(x)$

Gli schedule S_3 ed S_4 sono **view-equivalenti** e quindi S_3 è **view-serializzabile** in quanto S_4 è seriale. Gli schedule S_5 ed S_6 sono **view-equivalenti** ma non sono **view-serializzabili** in quanto nessuno dei due è seriale.

Si ha che se gli schedule presentano anomalie di perdita di aggiornamento, letture inconsistenti o di aggiornamento fantasma non sono **view-serializzabili**.

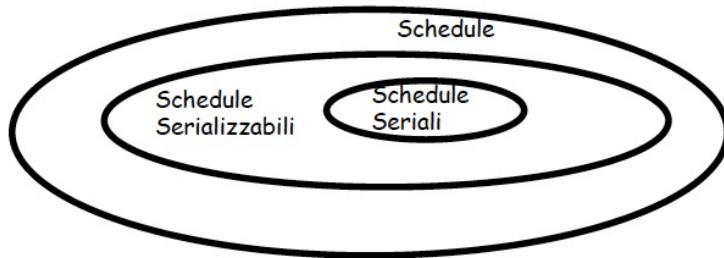
Un esempio sono i seguenti schedule:

S7: r1(x) r2(x) w1(x) w2(x) (perdita di aggiornamento)

S8 : r1(x) r2(x) w2(x) r1(x) (lettura inconsistenti)

S9 : r1(x) r1(y) r2(z) r2(y) w2(y) w2(z) r1(z) (aggiornamento fantasma)

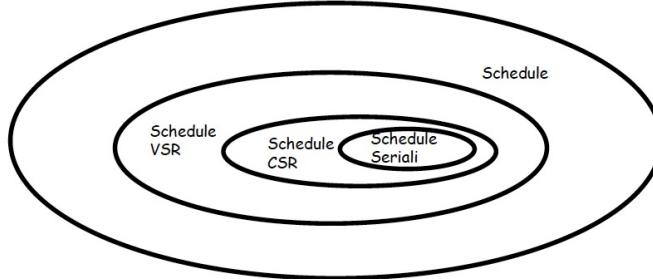
Verificare la view-equivalenza ha complessità polinomiale in quanto la verifica si traduce in una semplice verifica delle relazioni "legge da" e della scrittura finale, mentre verificare la view-serializzabilità ha una complessità esponenziale in quanto prevede il confronto con tutti gli schedule seriali. Questo non è quindi utilizzabile in pratica.



Schedule conflict - seriali sovrapposti

→ È necessario quindi restringere l'insieme della view-serializzabilità, introducendo il concetto di conflitto. Date due operazioni a_i e a_j , ovviamente con i diverso da j , si dice che esse sono in conflitto se esse lavorano sullo stesso oggetto ed almeno una di esse è un'operazione di scrittura. Si possono quindi verificare conflitti read-write (rw o wr) oppure conflitti write-write (ww). Si dice quindi che due schedule S_i e S_j sono conflict equivalenti se le operazioni in conflitto sono le stesse e appaiono nello stesso ordine nei due schedule. Si dirà quindi che uno schedule è conflict-serializzabile se esiste uno schedule seriale ad esso conflict-equivalente.

→ L'insieme degli schedule conflict-serializzabili si indica con CSR. Si ha che l'insieme CSR è un sottoinsieme proprio di VSR e ogni schedule conflict-serializzabile è anche view-serializzabile, mentre non vale il contrario (condizione sufficiente ma non necessaria). Per verificare che uno schedule sia conflict-serializzabile utilizziamo i grafi. Per ciascuno schedule, possiamo costruire il grafo nel modo che segue: Si costruisce un nodo per ciascuna transazione t_i che compare all'interno dello schedule e tracciamo un arco da t_i a t_j se esistono almeno due azioni a_i e a_j , tali che a_i precede a_j , in conflitto tra loro. Si ha che uno schedule è conflict-serializzabile se e solo se il grafo appena costruito è aciclico. La complessità adesso è lineare, ma questo risulta ancora inapplicabile in pratica in quanto il grafo verrà modificato dinamicamente durante l'utilizzo del database e comunque il processo è troppo dispendioso in termini di risorse e non è infine applicabile su database distribuiti.

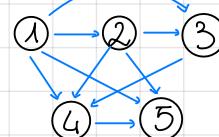


Un esempio

$S = r_1(x) w_2(x) r_3(x) r_1(y) w_2(y) r_1(v) w_3(v) r_4(v) w_4(y) w_5(y)$

Divisione per variabili :

x	y	z
r_1	r_1	r_1
w_2	w_2	w_3
r_3	w_4	r_4
		w_5



Il lock

→ È necessario quindi introdurre nuovi metodi per verificare la serializzabilità degli schedule. Un modo possibile è quello poi effettivamente utilizzato del locking. Il locking prevede che tutte le operazioni di lettura e scrittura debbano essere protette con le primitive di `r_lock`, `w_lock` ed `unlock`. Essendo che lo scheduler gestisce queste primitive, di fatto esso diventa un lock manager. In particolare, vi sono alcuni vincoli che devono essere verificati nell'esecuzione di queste transazioni. Innanzitutto, ogni operazione di lettura deve essere preceduta da `r_lock` e seguita da `unlock`. Vi è un lock condiviso in quanto possono essere presenti più lock in lettura per lo stesso dato. Per quanto riguarda le operazioni di scrittura invece le operazioni devono essere precedute da `w_lock` e seguite da `unlock`. Non è possibile che siano presenti più lock di questo tipo su un dato quindi il lock è esclusivo. Se una transazione rispetta questi vincoli si dice che è ben formata rispetto al locking. Quando una transazione deve leggere e scrivere contemporaneamente può passare da un lock di tipo condiviso (lettura) ad un lock di tipo esclusivo (scrittura), effettuando un lock upgrade. Il gestore della concorrenza gestisce i lock per le varie transazioni. Si ha che una transazione acquisisce il lock quando gli

viene concesso dal gestore, mentre si dice che lo rilascia quando viene fatto un unlock. Quando non viene concesso il lock, la transazione viene messa in attesa. I lock concessi vengono posti all'interno della tabella dei lock. Viene quindi costruita una tabella dei conflitti in cui le righe rappresentano le richieste, mentre le colonne rappresentano lo stato di una determinata risorsa. La richiesta di un lock in lettura ha risultato positivo anche se la risorsa è già bloccata in lettura, trattandosi di un lock condiviso, mentre le richieste di unlock possono non avere successo se la risorsa è bloccata da altre transazioni. Si ha quindi un contatore per tener traccia delle transazioni che usano la risorsa.

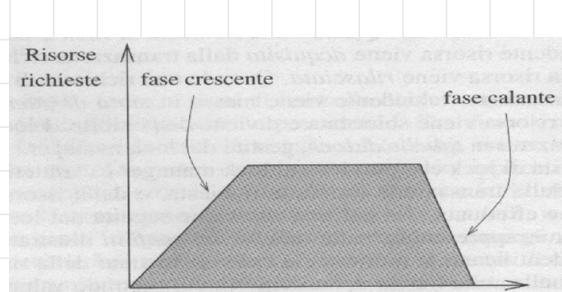
La tavola dei conflitti → Richiesta

	free	r_locked	w_locked
r_lock	OK / r_locked	OK / r_locked	NO / w_locked
w_lock	OK / w_locked	NO / r_locked	NO / w_locked
unlock	error	OK / depends (1)	OK / free

Stato della risorsa

2PL e 2PL stretto

Per garantire che gli schedule siano serializzabili è necessario però introdurre la tecnica del two-phase locking (2PL). Esso prevede che una transazione, una volta rilasciato un lock, non possa acquisirne altri. Si può quindi distinguere una fase crescente, nella quale una transazione acquisisce i lock, ed una fase calante, durante la quale li rilascia. Il passaggio da r_lock a w_lock costituisce un incremento di lock nella fase crescente. La classe 2PL contiene tutti schedule serializzabili ed è un sottoinsieme proprio della classe CSR.



Un esempio:

- begin (T1)
- w1_lock(B);
- r1(B);
- B:=B-50;
- w1(B);
- w1_lock(A);
- r1(A);
- unlock(B);
- A:=A+50;
- w1(A);
- unlock(A);
- commit

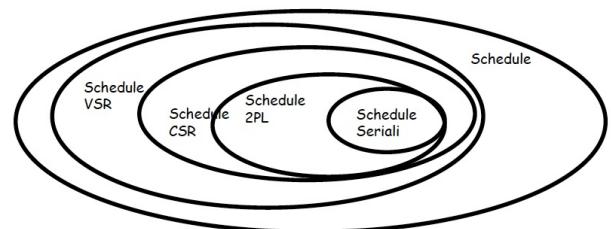
- begin (T1)
- w1_lock(B);
- r1(B);
- B:=B-50;
- w1(B);
- w1_lock(A);
- r1(A);
- A:=A+50;
- w1(A);
- unlock(A);
- commit

Il 2PL non elimina l'anomalia della lettura sporca in quanto un'altra transazione può aver già visto le risorse rilasciate dalla prima transazione che poi ha abortito. C'è quindi bisogno dell'abort in cascata. Un nuovo problema sorto con il 2PL sono le situazioni di deadlock. Se viene bloccata la transazione 1 perché aspetta x dalla transazione 2 e vice-versa, allora queste si bloccano e non possono essere sbloccate, generando un ciclo nelle attese. Il primo problema può essere risolto con il 2PL stretto, nel quale i lock vengono rilasciati soltanto dopo il commit/abort. Il secondo problema è invece di minore rilevanza rispetto al primo in quanto la probabilità che esso accada è $1/n^2$, dove n rappresenta le tuple su cui agisce la transazione. Anche per questa anomalia esistono delle tecniche che ne permettono la risoluzione.

Un esempio:

- begin (T1)
- w1_lock(B);
- r1(B);
- B:=B-50;
- w1(B);
- w1_lock(A);
- r1(A);
- A:=A+50;
- w1(A);
- commit
- unlock(B);
- unlock(A);

- begin (T1)
- w1_lock(B);
- r1(B);
- B:=B-50;
- w1(B);
- w1_lock(A);
- r1(A);
- unlock(B);
- A:=A+50;
- w1(A);
- unlock(A);
- commit

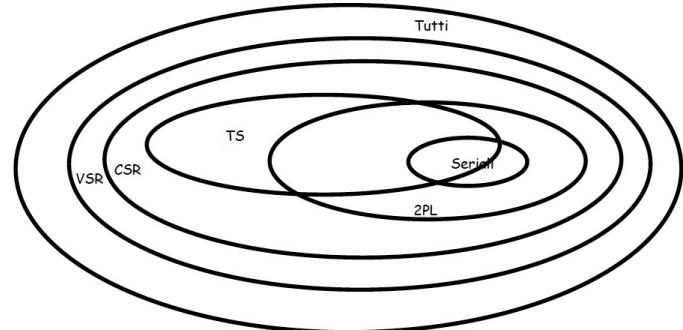


Concordanza con timestamp

Vi è una tecnica alternativa al 2PL che genera schedule **conflict-serializzabili**. Questa tecnica si basa sui **timestamp**. Per ogni transazione viene assegnato un timestamp nel quale viene indicato il momento in cui essa inizia, utilizzando l'orologio del sistema. Lo **schedule seriale equivalente** dovrà avere quindi le etichette ordinate nello stesso modo dell'altro schedule. Lo scheduler ha due contatori per ogni risorsa, un contatore **RTM(x)** e **WTM(x)**. Il primo rappresenta il **timestamp** della transazione con t più grande che ha letto x, mentre il secondo rappresenta il **timestamp** dell'ultima transazione che ha scritto su x. Allo scheduler può essere richiesto sia di leggere che di scrivere tramite le primitive **rt(x)** e **wt(x)**, dove t rappresentano i timestamp. Per quanto riguarda **rt(x)** o **read(x,ts)** si ha che se **ts < WTM(x)** allora la richiesta viene uccisa, altrimenti viene accolta e **RTM(x)** assume il valore maggiore tra **RTM(x)** e **ts**. Per quanto riguarda **wt(x)** o **write(x,ts)** se **ts < WTM(x)** o di **RTM(x)** la richiesta viene respinta, altrimenti **WTM(x)** assume il valore di **ts**. In pratica, una transazione non può leggere o scrivere un dato scritto da una transazione con timestamp superiore, e non può scrivere su un dato che è già stato letto da una transazione con timestamp superiore. L'ordine in cui le transazioni dichiarano il loro **begin** è quello di **assegnazione di timestamp**. Il risultato è **serializzabile** se equivalente a quell'ordine di timestamp. Con questo metodo di gestione, molte transazioni vengono uccise e devono ripartire, ed in generale sono più costose dell'attesa. A causa di questo molte transazioni vengono trasferite in memoria di massa soltanto dopo il commit. Vi sono due modifiche che possono essere apportate per risolvere il problema del grande numero di transazioni uccise. La prima modifica che può essere apportata è la regola di Thomas, per la quale, se si verifica, per **write**, che **ts < WTM(x)**, allora viene soltanto ignorata la scrittura. La seconda modifica è quella delle **multiversioni**, che permette di mantenere diverse versioni dei dati del database e indirizzare le scritture e le letture dei dati alla versione giusta, in base al timestamp. In questo modo abbiamo risolto i **deadlock** ma in generale la probabilità di deadlock è molto minore di quella di conflitti in TS, per cui si preferisce il 2PL.

Un esempio

Risposta	Nuovo Valore
read(x,1) Ok	RTM(x)=1
write(x,1) Ok	WTM(x)=1
read(z,2) Ok	RTM(z)=2
read(y,1) Ok	RTM(y)=1
write(y,1) Ok	WTM(y)=1
read(x,2) Ok	RTM(x)=2
write(z,2) Ok	WTM(z)=2



Problemi principali di entrambe le tecniche

Abbiamo visto che, nell'esecuzione delle transazioni, possano formarsi degli stalli. Uno stallo è un ciclo nel grafo delle attese. Le transazioni che non possono essere immediatamente soddisfatte vengono infatti poste all'interno di una coda ed ordinate per timestamp. Quando una risorsa si libera, il lock manager concede la risorsa alla transazione che è in coda da più tempo. La probabilità che due transazioni vadano in conflitto è pari a $k*m/n$, dove k è il numero di transazioni sul sistema, m è il numero medio di risorse a cui una transazione deve accedere ed n è il numero di oggetti presenti sulla base di dati. Le tabelle di attesa vengono poste in memoria centrale in quanto l'accesso ad esse è molto frequente, e contengono il numero di processi in attesa di una risorsa. Per gestire questa coda si usa il timeout. Il timeout è un tempo prefissato che stabilisce se una transazione debba essere abortita o meno. Se la transazione non ottiene la risorsa nel tempo stabilito, essa viene abortita. Un timeout non può essere né troppo alto né troppo basso in quanto bisogna cercare di uccidere meno transazioni possibili. Le transazioni uccise sono quelle che detengono le risorse richieste, in modo da farle rilasciare, oppure quelle che creano i cicli. Nel primo caso si parla di strategia preemptive. Un metodo efficace per determinare le transazioni da uccidere è il seguente: se più di una ha lock su una variabile viene uccisa quella che ha lavorato di meno. Si crea però in alcuni casi il problema di starvation. E' possibile infatti che sia sempre scelta la stessa transazione come bersaglio, non permettendole mai di fare lavoro. Per migliorare questo metodo, le transazioni mantengono sempre lo stesso timestamp e viene data la priorità a quelle più vecchie. I deadlock frequenti si verificano quando due transazioni cercano di incrementare il livello di lock. Per prevenire che questo accada è stato introdotto il lock di tipo update, che è compatibile con i lock di lettura. Un altro metodo utilizzato è quello del rilevamento di blocchi critici, che non pone vincoli al funzionamento, ma ispeziona di tanto in tanto l'esecuzione del sistema e vede se vi sono cicli in un grafo. E' possibile prevedere la possibilità di deadlock e, se alta, utilizzare il sistema del timestamp.

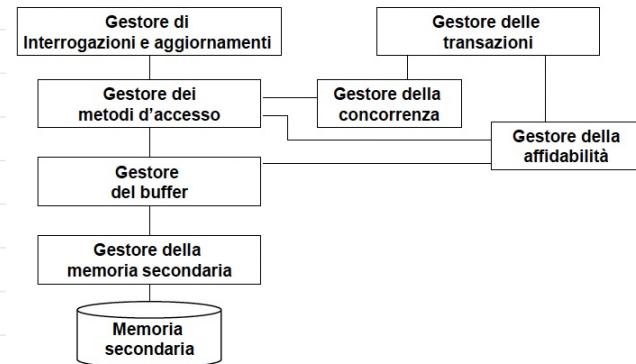
La struttura fisica

• Il DBMS ed il file-system

Un DBMS deve essere efficiente ed efficace e deve gestire molti dati che vengono forniti in ingresso. Questi dati sono grandi, persistenti e condivisi e deve gestirli garantendo affidabilità e privatezza. Oltre a questo, il DBMS deve gestire questi dati in modo efficiente. Per quanto riguarda la memoria secondaria, il DBMS si appoggia al filesystem del sistema operativo e le primitive per creare e gestire i file sono quelle del sistema operativo. Il DBMS gestisce il database come un unico grande file che costituisce uno spazio in memoria. Il file è organizzato in un insieme di record diviso in blocchi con all'interno diverse relazioni. Chiaramente le operazioni aumentano di velocità se i dati utilizzati per svolgerle sono presenti nello stesso blocco oppure in blocchi adiacenti. I dati in memoria secondaria, prima che vengano fatte operazioni su di essi, devono essere portati nel buffer e poi in memoria principale. Le uniche operazioni possibili in memoria secondaria sono lettura e scrittura di blocchi.

• La gestione delle pagine del buffer

Il buffer è invece organizzato in pagine. Un blocco viene trasferito in una pagina ed in esso sono presenti varie tuple di una relazione. Esse sono organizzate in blocchi contigui. La dimensione del blocco dipende dal filesystem. In generale, in una pagina sono mantenute sia informazioni riguardanti i dati veri e propri, ma anche informazioni di controllo. Essendo che una pagina corrisponde ad un blocco in memoria di massa, sono presenti in ogni pagina un block header ed un block trailer che mantengono le informazioni di controllo del filesystem. Vi sono poi i page header e trailer che contengono l'informazione di controllo della struttura fisica. In questi vengono mantenuti l'identificatore dell'oggetto presente nella pagina, memoria libera, numero di dati contenuti e puntatori a pagine precedenti o successive. Una pagina ha poi un dizionario che contiene i puntatori ai dati utili contenuti in essa. Si ha infine un bit di parità che permette di verificare che l'informazione contenuta sia valida. In questo sistema di database relazionale la lunghezza delle tuple è fissa. Se questa è variabile, il dizionario mantiene gli offset, ovvero le distanze di ciascuna tupla dalla parte utile ed i valori dei campi presenti rispetto all'inizio della tupla. Si parla di fattore di blocco per indicare quante tuple possono essere poste all'interno di un blocco. Esso è dato dal rapporto tra la dimensione di un blocco e la dimensione media dei record. Questo costituisce un limite inferiore in quanto potrebbe esserci spazio inutilizzato, che può però essere utilizzato anche per altre relazioni. Per quanto riguarda l'inserimento o l'aggiornamento non si ha riorganizzazione se vi è spazio sufficiente e lo spazio allocato è contiguo. Si inserisce invece nuovo spazio se questo non è sufficiente o riorganizzato l'attuale se questo non è contiguo. La cancellazione è sempre possibile e prevede che la tupla sia marcata come non valida. Gli accessi alle tuple o ai campi di una tupla sono dati dalla chiave o dall'offset.



• Strutture primarie e secondarie sequenziali

Le strutture primarie possono avere un'organizzazione sequenziale, ovvero possono essere ordinate in modo contiguo, ad esempio in ordine di inserimento. Questo ordinamento è soltanto fisico ma non logico. Possono poi essere ordinate in modo logico ma non in modo fisico, facilitando in questo modo alcune operazioni. Possono poi essere ordinate secondo un accesso calcolato, nel quale l'indice di un'informazione può essere calcolata. Le strutture sequenziali prevedono che le tuple siano poste in blocchi di memoria contigui e si possono avere 3 tipologie di organizzazione. Il primo metodo è quello della struttura disordinata. Per quanto riguarda questo metodo si ha un file heap disordinato e le informazioni nuove possono essere poste in modo casuale, per esempio al posto di un dato estratto oppure in coda alle informazioni già presenti. Se vi sono troppe posizioni vuote è necessario fare un compattamento. Questo metodo è conveniente per le operazioni di inserimento, modifica ed eliminazione. Per le prime è sufficiente memorizzare il puntatore all'ultimo elemento inserito, mentre per le ultime è sufficiente invece segnare come eliminati i dati. Non è invece molto efficiente per le operazioni di ricerca, in quanto questa ha tempo di esecuzione lineare.

- La seconda tipologia di strutture sequenziali è quella ad array. Questa è possibile solo se le tuple di una tabella hanno dimensione fissa. Per questa struttura vengono assegnati **n blocchi ad un file** e ciascuno di essi ha **m posizioni disponibili**, creando di fatto un array di $m \times n$ elementi. Ciascuna tupla ha un proprio indice **i** e possono essere fatte operazioni di inserimento nelle posizioni libere oppure operazioni di cancellazione che liberano queste posizioni.
- La terza struttura sequenziale è quella ordinata. Le tuple in questo caso vengono ordinate secondo un campo detto chiave. Questa tipologia di struttura sequenziale viene usata parallelamente agli indici. Sulle strutture ordinate è possibile fare ricerche più efficaci di tempo logaritmico, specialmente se le ricerche riguardano intervalli di valori, ma è invece più complicato mantenere l'ordinamento.

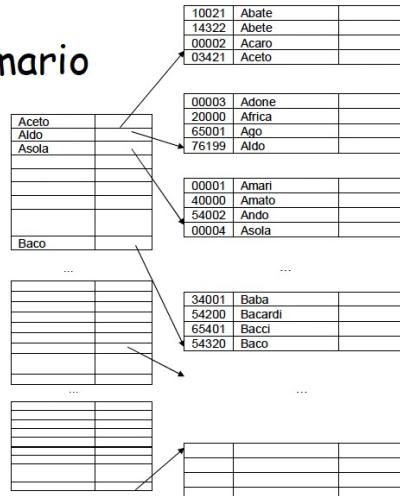
→ L'ultimo metodo riguarda le tabelle hash. Questo metodo prevede che venga utilizzata una funzione hash che trasforma il valore della chiave in indici di un array.

Un esempio di funzione hash è quella che produce l'indirizzo dividendo la chiave per la dimensione della tabella. In questo modo si ha **accesso diretto**, ma può verificarsi il problema delle collisioni quando due elementi differenti hanno lo stesso indirizzo hash. Questo crea un problema in quanto è poi necessario **ricercare l'elemento corretto all'interno dell'insieme delle collisioni**. La complessità di questa ricerca dipende strettamente dalla grandezza dell'insieme delle collisioni, che diminuisce con l'aumento della dimensione della struttura. Per la memorizzazione tramite tabelle hash si utilizza un fattore fondamentale che è il **fattore di riempimento**. Esso indica lo spazio fisico mediamente utilizzato. Se T è il numero di tuple previsto per il file, F è il fattore di blocco ed f è il fattore di riempimento, il **numero di blocchi** è l'intero immediatamente superiore a $B = T/(f * F)$. La funzione hash restituisce un numero tra 0 e $B-1$, che costituisce il **numero del blocco** nel quale è presente l'informazione. Se si verificano collisioni, si utilizza lo **stesso blocco** per la **memorizzazione di informazioni fino ad esaurimento dello spazio**. Una volta esaurito ne viene allocato altro e può dar luogo a **catene di overflow**. La loro lunghezza è direttamente proporzionale al fattore di riempimento ma inversamente proporzionale al fattore di blocco. Questa tipologia è utile nel caso di **accessi puntuali**, ovvero tramite un singolo valore della chiave in quanto questa operazione in assenza di collisioni ha **complessità costante**. Non è invece **conveniente** per l'accesso tramite **intervalli di valori** in quanto le funzioni hash generalmente tendono a spargagliarli. Non si ha **nessun vantaggio** nemmeno per valori diversi da quelli chiave ed inoltre la struttura ha lo **svantaggio** di essere **scarsamente dinamica**, specialmente per quanto riguarda l'inserimento. Quando il file diventa **molto grande** è infatti comune dover riorganizzarlo **creando anche eventualmente una nuova funzione hash**.

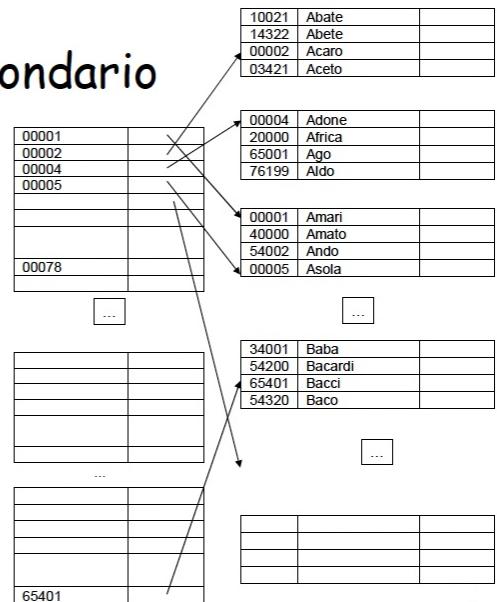
Gli indici

→ Fino ad ora abbiamo analizzato tecniche di memorizzazione sequenziale. Per quanto riguarda la memorizzazione **non sequenziale**, si utilizzano strutture ad **albero**, alle quali si accede tramite i **puntatori**. Le strutture non ordinate permettono sia **accessi puntuali**, sia accessi ad **intervalli di valori**. Esse possono essere utilizzate sia per **strutture primarie** che per **strutture secondarie**. Una tipologia di struttura secondaria sono gli **indici**, che possono essere **primari o secondari**. Per quanto riguarda gli indici secondari, dato un file f con un campo chiave k , un indice secondario è un altro file che ha record composti da due campi contenenti uno il **valore della chiave k** e l'altro contenente gli **indirizzi fisici** che hanno quel valore di chiave. Gli indici secondari permettono di **recuperare dati** riguardanti interrogazioni che si basano sulla chiave k . Se l'indice contiene anche i dati oppure è ordinato sullo stesso campo su cui è definito l'indice esso è detto **primario**. Un file può avere un solo indice primario e più indici secondari. Un file ordinato sequenzialmente o hash secondo un campo che non è quello dell'indice non può avere indice primario. Gli indici possono contenere anche gli **offset** all'interno dei blocchi. La seconda soluzione permette di rendere più efficienti alcune interrogazioni complesse in quanto non è richiesto l'accesso al file contenente i dati se non alla fine. Gli indici possono essere **densi o sparsi**. Nel caso di indici sparsi possono esserci **valori della chiave che non sono presenti** all'interno dell'indice. Gli indici sparsi sono compatibili soltanto con gli indici primari data l'**organizzazione dei file in blocchi contigui**. Gli indici secondari possono invece avere dei **valori della chiave** presenti in **blocchi diversi**, quindi gli indici secondari devono essere necessariamente **densi**. Gli indici permettono **ricerche efficienti**, anche data la **minore dimensione dell'indice rispetto al file**. Essi rendono inoltre più efficienti le ricerche su **intervalli** e le **scansioni sequenziali ordinate**. Essi sono realizzati tramite strutture ad albero.

Indice primario



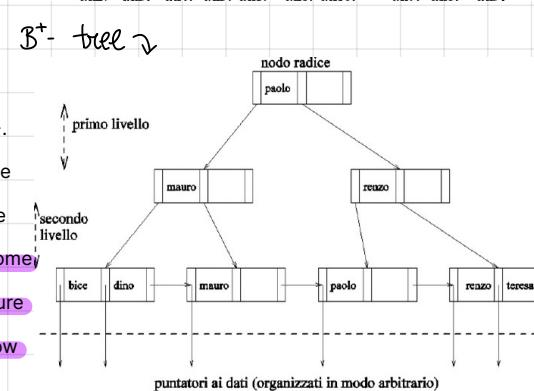
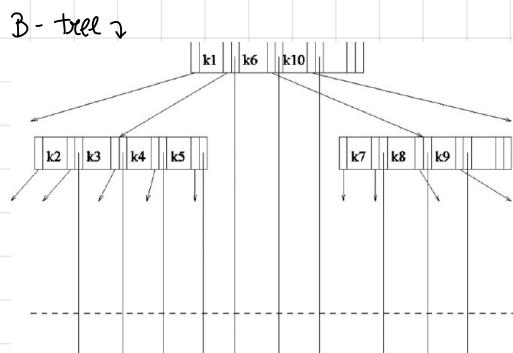
Indice secondario



→ Per quanto riguarda la struttura ad albero utilizzata, ogni nodo corrisponde ad una pagina oppure ad un blocco e sono legati da puntatori che collegano tra loro le pagine. Ciascun nodo ha un numero elevato di discendenti, che permette di creare alberi con un numero limitato di livelli. È conveniente che gli alberi siano bilanciati in modo che il tempo per accedere ai dati sia costante e che sia pari alla profondità dell'albero. Ciascun nodo presenta una precisa struttura con F valori ordinati di chiave. Ciascuna chiave K_i , con i compreso tra 1 ed F, è seguita da un puntatore P_i , mentre la chiave K_1 è preceduta dal puntatore P_0 . Quest'ultimo permette di accedere al sottoalbero che permette di accedere ai record con chiavi minori di K_1 . Allo stesso modo P_F permette di accedere al sottoalbero che contiene le chiavi con valore maggiore o uguale a K_F . I puntatori intermedi permettono di accedere ai sottoalberi che contengono i valori di chiave nell'intervallo $[K_i, K_{i+1}]$. Ogni nodo ha F valori di chiave ed $F+1$ (fan-out) puntatori, dove il valore F dipende dall'ampiezza della pagina, dallo spazio occupato dai valori chiave e dai puntatori della parte utile della pagina. Il valore F scelto è il massimo possibile, in modo da ridurre il numero di livelli dell'albero. In ogni caso, questo valore non viene mai raggiunto in modo da garantire più flessibilità. Per effettuare la ricerca in un albero si utilizza una primitiva che parte dalla radice e poi prosegue a seconda del valore assunto dalla chiave fino ad arrivare ai nodi foglia. Questi possono essere organizzati in due modi: nel caso di indici primari essi contengono dati e quindi si ha una struttura index-sequential e realizza una struttura ordinata in base al suo indice primario, mentre nel caso di indici secondari il posizionamento del file è casuale e quindi si ha una struttura detta indiretta che permette di utilizzare qualsiasi tecnica di indirizzamento primaria. Tutto ciò perché i nodi foglia contengono puntatori ai blocchi della base di dati con quel valore chiave. Per quanto riguarda il numero di blocchi per un indice primario si avrà $N_b = T^*R/B$. Per quanto riguarda il numero di livelli si avrà il logaritmo di N_b . Se l'indice primario è separato dal file, N_b rimane lo stesso, mentre invece, avendo che per ogni foglia si ha un record per ogni blocco del file, il numero di foglie è N_b diviso il fan-out, quindi ha un livello in meno di prima. Per un indice denso il numero di blocchi è dato da $N_d = T*(K+P)/B$, mentre per un indice sparso il numero di blocchi è dato da $N_s = N_b * (K+P)/B$. Abbiamo che T è il numero di record nel file, B è la dimensione dei blocchi, R è la lunghezza dei record, K è la lunghezza del campo chiave e P la lunghezza degli indirizzi. Nella memoria secondaria sono presenti più livelli di indici, sia per gli indici primari che per gli indici secondari. L'indice su strutture ordinate ha un comportamento poco flessibile quando c'è bisogno di flessibilità. In generale gli indici utilizzati sono dinamici e multilivello. Per questo vengono utilizzati gli alberi binari. Gli inserimenti e le cancellazioni prevedono che anche gli indici vengano aggiornati di conseguenza, eccetto nel caso in cui l'inserimento possa essere fatto in una foglia. Se questo non accade, deve essere fatta un'operazione di split, che divide la vecchia e la nuova informazione in modo da poterla assegnare equamente a due foglie. La split può propagarsi anche ai livelli superiori generando una catena, data la necessità di aggiungere puntatori ai livelli superiori. La cancellazione consiste nel marcare lo spazio come invalido. Nel caso di un indice sparso inoltre viene spostato il valore successivo della chiave e viene messo al posto del valore cancellato, mentre se in una foglia vi sono informazioni che sono scarsamente utilizzate, esse possono essere unite con la merge in un'unica pagina. Anche la merge provoca una modifica (diminuzione) dei puntatori ai livelli superiori e questo può quindi generare ancora una catena come nel caso della split. Questa organizzazione permette un riempimento medio dei nodi superiore al 50%.

B-Tree e B⁺-tree

→ Per quanto riguarda le strutture ad albero, ne esistono due tipologie, le B-Tree e le B+-tree. Per quanto riguarda la tipologia B+, abbiamo che le foglie di questa tipologia sono collegate tra di loro nell'ordine imposto dalla chiave, permettendo l'esecuzione più semplice di richieste riguardanti insiemi di valori, in quanto basterà accedere al primo valore e poi scorrere in avanti. Nel caso index sequential si avranno subito i dati richiesti come risposta, mentre nel caso indiretto sarà necessario accedere ai dati tramite i puntatori. Questa struttura permette inoltre anche la scansione ordinata in base ai valori di chiave dell'intero file. La struttura B-tree non prevede il collegamento tra le foglie ma per ottimizzare le operazioni vengono invece utilizzati due puntatori nei nodi intermedi. Il primo puntatore permette di puntare al blocco contenente K_i e di interrompere la ricerca, mentre il secondo puntatore prosegue la ricerca nel sottoalbero contenente i valori tra K_i e K_{i+1} . P_0 individua il sottoalbero contenente i valori inferiori a K_1 , mentre P_F quelli superiori a K_F . Essendo che ciascun valore è presente univocamente nell'albero, questa tecnica è molto efficiente e permette di risparmiare spazio rispetto alla struttura B+. L'efficienza di questa struttura è garantita da alcune tecniche di compressione e dal fatto che le pagine contenenti i primi livelli risiedono nel buffer. Nei gestori di database classici vi sono vari tipi di strutture primarie e secondarie. Oracle usa una struttura primaria disordinata con accesso hash ed un b-tree come struttura secondaria. Gli indici sono di vario tipo. In SQL server vi sono indici con B-tree densi e strutture primarie con file heap o B-tree sparsi. È possibile vedere che effetto ha l'indice tramite il comando show plan. Questo comando permette di valutare le prestazioni del sistema e modificare l'indice utilizzato in modo da aumentare l'efficienza. Con questo approccio iterativo è possibile migliorare l'efficienza del sistema. Alcune operazioni vengono fatte automaticamente, come per esempio definire gli indici sulle chiavi primarie. Altri indici su dati che per esempio fanno join di frequente possono essere definiti dall'utente. Il comando utilizzato per definire l'indice è il seguente: `create [unique] index NomeIndice on NomeTabella (ListaAttributi)`. Per eliminare l'indice si utilizza invece il comando `drop index NomeIndice`.



• Profili delle relazioni

→ Il gestore delle interrogazioni si occupa delle interrogazioni poste alla base di dati da parte dell'utente. Esso esegue prima dei controlli dal punto di vista della sintassi delle interrogazioni per poi passare a varie fasi di ottimizzazione del codice e terminare infine con l'esecuzione delle interrogazioni. Prima di essere eseguita, l'interrogazione viene tradotta in linguaggio algebrico, effettuando le varie fasi di ottimizzazione già viste inizialmente e poi esegue un'ottimizzazione che dipende dai metodi di accesso ai dati supportati ed anche dal modello dei costi che è stato assunto. Viene infine prodotto il codice che utilizza i metodi di accesso. Il codice può essere anche conservato nel caso in cui la stessa operazione venga ripetuta più volte e diventa non valido nel momento in cui la base di dati cambia molto. Il risultato dell'ottimizzazione è un'interrogazione SQL sotto forma di albero le cui foglie sono le tabelle, mentre i nodi intermedi sono operazioni in algebra relazionale. Per poter ottimizzare le interrogazioni il DBMS conserva informazioni riguardo a vari dati. Le più importanti sono le seguenti:

CARD (T): cardinalità di ogni tabella

SIZE (T): dimensione in byte di ogni tabella

SIZE (Aj, T): dimensione in byte di ogni attributo di T

VAL (Aj, T): numero di valori distinti assunti da ogni attributo di T

MIN (Aj, T) e MAX (Aj, T): minimo e massimo valore assunto dagli attributi di T.

L'ottimizzazione basata sui costi richiede molte considerazioni basate sulla statistica.

→ Per quanto riguarda una selezione T', i profili ottenuti sono i seguenti:

CARD(T') = $(1/\text{VAL}(A_j, T)) * \text{CARD}(T)$

SIZE(T') = SIZE(T)

VAL(Ai, T')=1

VAL(Aj, T')=col(CARD(T), VAL(Aj, T), CARD(T')), j!=i

MAX(Ai, T')=MIN(Ai, T')=v

MAX (Aj, T') e MIN (Aj, T') non differiscono da quelli di T

→ Per quanto riguarda le proiezioni, si analizza una proiezione T' con un insieme di attributi L

CARD(T')=MIN(CARD(T), $\prod_{i=1}^n \text{VAL}(A_i, T)$)

SIZE(T')= $\sum_{i=1}^n \text{SIZE}(A_i, T)$

VAL(Ai, T'), MAX(Ai, T'), MIN(Ai, T') rimangono invariati rispetto a T.

→ Per quanto riguarda i join, si considera un equi join Tj con $\text{VAL}(A, T')=\text{VAL}(B, T'')$

CARD(Tj)=(1/VAL(A, T'))*CARD(T')*CARD(T'')

SIZE(Tj)=SIZE(T')+SIZE(T'')

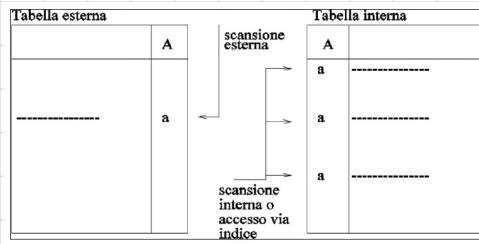
VAL(Ai, Tj), MAX (Ai, Tj), MIN(Ai, Tj) assumono gli stessi valori precedenti al join

• Ottimizzazione algebrica

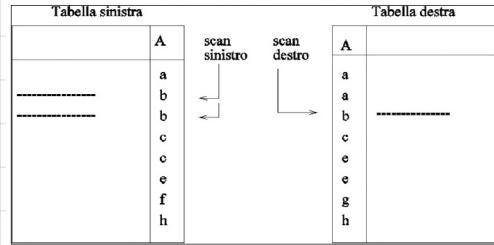
→ Le prime operazioni che vengono fatte per ottimizzare un'interrogazione sono trasformare le foglie in nodi che tengono conto delle strutture fisiche ed i nodi intermedi in operazioni di accesso. La prima operazione possibile è quella di scansione che opera su varie operazioni, quali proiezione, selezione e inserimenti, cancellazioni e modifiche. Per l'operazione di ricerca si ha una selezione che prevede una ricerca completa di ordine medio $n/2$. I blocchi per questa ricerca devono essere trasferiti all'interno del buffer in quanto è necessario fare i confronti. Se le tuple sono ordinate e la selezione è fatta sugli attributi ordinati, è possibile trasferire solo i blocchi su cui viene fatta la ricerca logaritmica e, tramite gli indici, possono essere trasferiti anche soltanto alcuni blocchi. Vi sono poi le operazioni di ordinamento, i quali algoritmi tengono conto sia delle caratteristiche della memoria secondaria che della disponibilità di buffer. Se questa aumenta, aumenta contestualmente anche l'efficienza degli algoritmi di ordinamento. Un metodo tipico di ricerca è molto simile al merge sort e ordina inizialmente porzioni di tabella pari alla dimensione del buffer e poi fonde tante porzioni quante sono le pagine disponibili nel buffer. Si ha accesso diretto quando sono presenti indici oppure una struttura ad hash. I primi favoriscono sia interrogazioni con condizioni puntuali sia quelle con condizioni ad intervalli. Quando vi sono più predicatori, viene scelto quello più selettivo per l'accesso diretto e poi viene valutato l'altro predicato per le pagine presenti nel buffer che soddisfano il primo predicato. Per predicatori di disgiunzione, quando uno non viene soddisfatto è richiesta subito una scansione lineare. Quando sono presenti indici poi è necessario eliminare quelle tuple che possono essere raggiunte mediante più indici.

Tecniche di JOIN

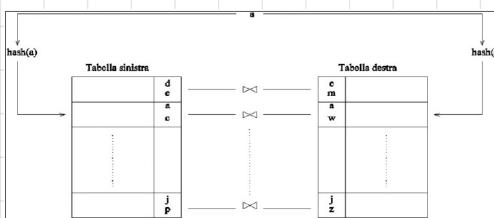
Il join è la parte più difficile da ottimizzare in quanto la cardinalità è data dal prodotto delle cardinalità degli operandi. Vi sono 3 tecniche di join. La prima tecnica è quella di **nested loop** che prevede di **selezionare una tabella come interna e l'altra come esterna**. Da quella **esterna**, si **ricerca l'attributo di join** e poi si effettua una **ricerca nella tabella interna** per gli attributi che **hanno lo stesso attributo di join**, utilizzando strutture hash o indici creati ad hoc sulla tabella interna.



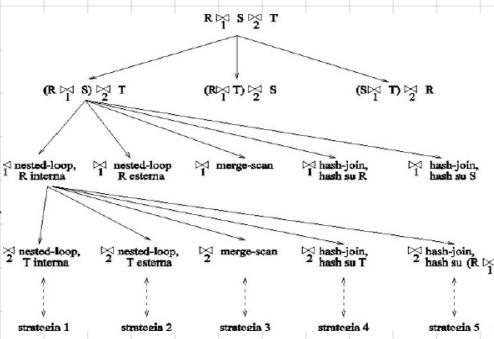
Il secondo metodo è il metodo di **merge scan**. Questo metodo richiede che le tabelle siano **ordinate** per gli attributi di join oppure che siano presenti indici. Questo metodo **non richiede di scansionare tutta la tabella interna** in quanto i valori sono ordinati. Questo metodo utilizza **scansioni parallele** che individuano l'elemento da cui partire e proseguono fin quando trovano un elemento diverso, arrestandosi in quanto le tabelle sono ordinate secondo l'attributo di join. Il **costo** di questo approccio è dato dall'**ordinamento** delle tabelle. Questo approccio è utilizzato nel caso di join naturale. Il blocco viene letto una sola volta ed i valori uguali si trovano nello stesso blocco e quindi stanno insieme in memoria. Il **costo** è dato dai **trasferimenti** e dall'**ordinamento**.



L'ultimo metodo utilizza le **funzioni hash** ed è utilizzato per **join naturali o equi-join**. La funzione hash permette di **partizionare le tuple di entrambe le relazioni**, facendo uso di **tabelle aggiuntive**, in particolare **applicando la stessa funzione hash ad entrambe le tabelle**. La **velocità del metodo aumenta in funzione della memoria utilizzata**. Una volta fatta la **divisione in partizioni** sarà necessario **fare tanti join** quante sono le **partizioni**. Per quanto riguarda la **memoria** si hanno **due fasi**. Nella **prima** vengono **trasferiti i dati per la lettura** e nella **seconda vengono scritte le tabelle aggiuntive**. Il DBMS crea un **albero di decisione**, le cui foglie rappresentano varie **strategie per implementare una query**, la **radice** è costituita dall'**operazione richiesta** ed i **nodi intermedi** rappresentano tutte le **alternative di decisione possibili**. Viene scelta la strategia che **richiede tempo minore**.



Per quanto riguarda il join, conviene fare prima il join che produce la tabella intermedia più piccola. In generale, si preferisce quindi che il **risultato intermedio sia il più piccolo possibile**. Per valutare il costo viene utilizzata la formula $C_{total} = C_{i/o} * n_i/o + C_{cpu} * n_{cpu}$, nella quale le **C** sono costanti, mentre **n** sono variabili e rappresentano il numero di operazioni di I/O ed il numero di istruzioni della CPU richieste per realizzarle. Viene quindi costruito l'albero e viene valutato il costo in termine di memoria e trasferimenti, per poi scegliere la **soluzione migliore**, che però non è necessariamente quella ottima da tutti i punti di vista. Viene scelta in generale la soluzione che ha lo stesso **ordine di grandezza della soluzione ottima** e vengono scartate tutte le altre.



La progettazione fisica

La progettazione fisica dipende dal **sistema di gestione utilizzato** e consiste nella scelta della struttura primaria per ciascuna relazione e nella definizione eventuale di indici secondari. Conviene quindi ottimizzare le operazioni di selezione e di join definendo una struttura hash oppure un indice. Di default viene definito un indice relativo alla chiave primaria, che il programmatore può decidere di rendere anche primario oppure di sostituire con una struttura hash. Possono inoltre essere costruiti dei **cluster multirelatazionali**. Il generale si cerca di ridurre il costo delle operazioni pesato con la loro frequenza. Per questo è utile creare degli indici aggiuntivi, che però è bene aggiungere solo se i vantaggi sono molti in quanto la loro aggiunta porta ad un peggioramento delle prestazioni delle operazioni di modifica.