UNIVERSITÀ DI PISA

Dipartimento di Ingegneria
Corso di Laurea Triennale in Ingegneria Informatica

Appunti

Basi di dati

Professori:
Prof. Pistolesi
Prof. Tonellotto

Autore:
Enea Passardi

Anno Accademico 2023/2024

Contents

Unimap

1. **Mer 28/02/2024 08:30-11:30 (3:0 h)** lezione: Presentazione del corso e struttura dell'esame. Introduzione al modello relazionale. Tabelle, record, attributi, chiavi. Linguaggi dichiarativi vs procedurali. Il DBMS Oracle MySQL. Linguaggio SQL. Formato del select statement. Esempi di query di base. Condizioni e connettivi logici. Condizioni su range (numerici e temporali). Il valore NULL. Gestione dei valori NULL. Condizioni che coinvolgono valori NULL. Rimozione dei duplicati e relativa complessità. Concetto di data in MySQL: UNIX timestamp. (Francesco Pistolesi)

2. **Gio 29/02/2024 10:30-13:30 (2:0 h)** lezione: Condizioni sulle date. Lassi di tempo e loro durata (date_diff). Sommare e sottrarre lassi di tempo. Esprimere condizioni basate sulla data corrente (current_date). Funzioni di aggregazione: conteggio di record, conteggio di valori diversi su attributo, somma, media, massimo e minimo. Introduzione alle query su più tabelle. Panoramica sui vari tipi di join. L'inner join. Funzionamento e modalità di generazione del result set. Considerazioni sulla cardinalità del result set. (Francesco Pistolesi)

3. **Ven 01/03/2024 14:30-16:30 (2:0 h)** lezione: Natural join. Outer join: come funziona e quando usarlo. Differenze fra inner join e outer join. Il ruolo dei valori NULL: quando eliminarli e quando non farlo. Self join. Riconoscere i vari tipi di join dal testo di una query. Query con join e condizioni sui record. Join su più di due tabelle. Ambiguità. Common table expression (CTE). Formulazione del codice di una query usando le CTE. (Francesco Pistolesi)

4. **Mer 06/03/2024 08:30-11:30 (3:0 h)** lezione: Introduzione al corso e informazioni logistiche. (Nicola Tonellotto)

5. **Gio 07/03/2024 10:30-13:30 (3:0 h)** lezione: Modello relazionale (Nicola Tonellotto)

6. **Ven 08/03/2024 14:30-16:30 (2:0 h)** lezione: Modello relazionale. Chiavi. Vincoli di integrità. (Nicola Tonellotto)

7. **Mer 13/03/2024 08:30-11:30 (3:0 h)** lezione: Algebra relazionale: operatori su insiemi. operatori su relazioni. join. (Nicola Tonellotto)

8. **Gio 14/03/2024 10:30-13:30 (3:0 h)** lezione: Algebra relazionale: equivalenza di espressioni e ottimizzazione algebrica. (Nicola Tonellotto)

9. **Ven 15/03/2024 14:30-16:30 (2:0 h)** esercitazione: Esercizi sul modello relazionale. (Nicola Tonellotto)

10. **Mer 20/03/2024 08:30-11:30 (3:0 h)** non tenuta: Indisponibilità del docente. (Francesco Pistolesi)

11. **Gio 21/03/2024 10:30-13:30 (3:0 h)** lezione: Calcolo relazionale: calcolo dei domini e calcolo delle tuple. (Nicola Tonellotto)

12. **Ven 22/03/2024 14:30-16:30 (2:0 h)** esercitazione: Esercizi su algebra e calcolo relazionale. (Nicola Tonellotto)

13. **Gio 04/04/2024 10:30-13:30 (3:0 h)** lezione: Introduzione alle subquery. Frammentazione di una query subquery e strategie di risoluzione. Noncorrelated subquery. Costruzione del result set. Condizioni in/not in. Equivalenza subquery-join. Passaggio alla versione join-equivalente. Annidamento multiplo. Subquery scalari. (Francesco Pistolesi)

14. **Ven 05/04/2024 14:30-16:30 (2:0 h)** esercitazione: Esercizi su algebra e calcolo relazionale. (Nicola Tonellotto)

15. **Mer 10/04/2024 08:30-11:30 (3:0 h)** lezione: Progettazione concettuale: ciclo di vita, analisi dei requisiti, modell ER, entità, relationship e attributi. (Nicola Tonellotto)

16. **Gio 11/04/2024 10:30-12:00 (1:30 h)** lezione: L'operatore di raggruppamento: quando si usa, come funziona e cosa restituisce. Sintassi del costrutto 'group by'. Attributi di raggruppamento. Applicazione delle funzioni di aggregazione. Condizioni sui gruppi. La having clause. Come riconoscere le condizioni sui record dalle condizioni sui gruppi. (Francesco Pistolesi)

17. **Gio 11/04/2024 12:00-13:30 (1:30 h)** esercitazione: Risoluzione ragionata di esercizi per casa. (Francesco Pistolesi)

18. **Ven 12/04/2024 14:30-16:30 (2:0 h)** esercitazione: Esercizi su algebra e calcolo relazionale. (Nicola Tonellotto)

19. **Mer 17/04/2024 08:30-11:30 (3:0 h)** lezione: Diagrammi E-R estesi. Design pattern per progettazione concettuale. (Nicola Tonellotto)

20. **Gio 18/04/2024 10:30-12:00 (1:30 h)** lezione: Correlated subquery. Come funzionano e a cosa servono. Costruzione del result set rispetto alla outer query. Operatori in/not in con subquery correlate. Le variabili di correlazione. Costrutto exists. Subquery correlate nel select. Considerazioni sull'efficienza delle subquery. L'operatore di divisione. Implementazione della divisione tramite doppia subquery not exists, e con raggruppamento e subquery di conteggio nella having clause. (Francesco Pistolesi)

21. **Gio 18/04/2024 12:00-13:30 (1:30 h)** esercitazione: Risoluzione ragionata di esercizi per casa. (Francesco Pistolesi)

22. **Ven 19/04/2024 14:30-16:30 (2:0 h)** esercitazione: Esercizi su progettazione concettuale. (Nicola Tonellotto)

23. **Mer 24/04/2024 08:30-11:30 (3:0 h)** lezione: Strategie di progettazione concettuale. Ristrutturazione di progetto. Progettazione logica. Traduzione da progetto concettuale a progetto relazionale. (Nicola Tonellotto)

24. **Ven 26/04/2024 14:30-16:30 (2:0 h)** esercitazione: Esercizi su progettazione concettuale e logica. (Nicola Tonellotto)

25. **Gio 02/05/2024 10:30-13:30 (3:0 h)** lezione: Stored procedure. Generalità, scopi, e vantaggi. L'istruzione create procedure. Parametri di ingresso e uscita. Definizione di statement e delimiter. Chiamata a stored procedure: il comando call. Variabili locali e user-defined. Scope delle variabili. Istruzioni condizionali if e case. Istruzioni iterative while, repeat, e loop. Istruzioni di salto leave e iterate. Processare result set. I cursori. Istruzioni open, fetch e close. Il ciclo di fetch. Handler. Gestione dell'evento not found tramite un handler di tipo continue. (Francesco Pistolesi)

26. **Ven 03/05/2024 14:30-16:30 (2:0 h)** lezione: Dipendenze funzionali: anomalie, definizione di DF, chiusura di attributi, chiusura di DF, implicazione logica e derivazione, teorema di correttezza e completezza. (Nicola Tonellotto)

27. **Mer 08/05/2024 08:30-10:30 (2:0 h)** lezione: Dipendenze funzionali: verifica di DF, chiavi e superchiavi, equivalenza, coperture minimali. Introduzione alla forma normale di Boyce-Codd. (Nicola Tonellotto)

28. **Mer 08/05/2024 10:30-11:30 (1:0 h)** esercitazione: Esercizi sulle dipendenze funzionali. (Nicola Tonellotto)

29. **Gio 09/05/2024 10:30-12:00 (1:30 h)** lezione: Data manipulation language. Istruzione insert. Inserimento di valori statici e ricavati. Default value e specifica dello schema. L'istruzione update. Problema della non riferibilità della target table nella condizione. Risoluzione tramite derived table, join anticipato e CTE. Istruzione delete. (Francesco Pistolesi)

30. **Gio 09/05/2024 12:00-13:30 (1:30 h)** esercitazione: Esercizi di ricapitolazione sulla parte dichiarativa. Risoluzione ragionata di un esercizio relativo alla divisione insiemistica, in tre alternative. (Francesco Pistolesi)

31. **Ven 10/05/2024 14:30-16:30 (2:0 h)** esercitazione: Esercizi sulle dipendenze funzionali. (Nicola Tonellotto)

32. **Mer 15/05/2024 08:30-11:30 (3:0 h)** lezione: Forma normale di Boyce-Codd, terza forma normale. (Nicola Tonellotto)

33. **Mer 15/05/2024 16:30-17:30 (1:0 h)** lezione: Trigger. Sintassi dell'istruzione create trigger. Trigger after e before. Esempio di aggiornamento di attributi ridondanti tramite trigger. [Lezione in modalità ibrida: in presenza + streaming Teams] (Francesco Pistolesi)

34. **Mer 15/05/2024 17:30-18:30 (1:0 h)** esercitazione: Risoluzione ragionata di un esercizio d'esame. [Esercitazione in modalità ibrida: in presenza + streaming Teams] (Francesco Pistolesi)

35. **Gio 16/05/2024 10:30-12:00 (1:30 h)** lezione: Introduzione alle materialized view (MV). Concetti di base e ottimizzazione delle performance per il recupero delle informazioni. Com'è fatta una MV. Definizione dello schema. Build tramite stored procedure. Mantenimento in sync con i raw data. Full refresh di tipo deferred e on-demand. (Francesco Pistolesi)

36. **Gio 16/05/2024 12:00-13:30 (1:30 h)** esercitazione: Esercizio sulle materialized view. (Francesco Pistolesi)

37. **Ven 17/05/2024 14:30-16:30 (2:0 h)** esercitazione: Esercizi sulle forme normali. (Nicola Tonellotto)

38. **Mer 22/05/2024 08:30-11:30 (3:0 h)** lezione: Tecnologie delle basi di dati: transazioni e gestore della concorrenza. Schedule seriali, View-equivalenza, conflict-equivalenza. Lock, algoritmo 2PL. (Nicola Tonellotto)

39. **Mer 22/05/2024 16:30-17:30 (1:0 h)** lezione: Incremental refresh di una materialized view. Log table. Definizione dello schema della log table. Esempi. Trigger di push. Codice per l'incremental refresh di tipo deferred e on demand (partial e complete). (Francesco Pistolesi)

40. **Mer 22/05/2024 17:30-18:30 (1:0 h)** esercitazione: Esercizi sulle materialized view. (Francesco Pistolesi)

41. **Gio 23/05/2024 10:30-13:30 (3:0 h)** lezione: Window functions. Concetto di partition. Operatori aggregati calcolati su partition. Clausola over. Impiego di partition by e order by. Funzioni rownumber, rank e dense_rank. Reperimento di valori non aggregati all'interno della partizione: le funzioni lead e lag. Concetto di frame. Window functions su frame. Il default frame. Le funzioni first_value e last_value. Definizione di frame personalizzati tramite il costrutto rows. (Francesco Pistolesi)

42. **Ven 24/05/2024 14:30-16:30 (2:0 h)** esercitazione: Esercizi di ricapitolazione, considerazioni conclusive e chiarimenti sulle modalità d'esame. (Francesco Pistolesi)

43. **Mer 29/05/2024 08:30-11:30 (3:0 h)** lezione: Gestore dell'affidabilità: log, dump, checkpoint. Scrittura su memorie secondarie. Ripresa a caldo e ripresa a freddo (Nicola Tonellotto)

44. **Ven 31/05/2024 14:30-16:30 (2:0 h)** esercitazione: Sessione di domande e risposte in preparazione dell'esame. (Nicola Tonellotto)

Introduzione alle basi di dati

Sistema informativo

Si definisce come ***sistema organizzativo*** un'insieme di *regole* e *risorse* coordinate tra loro atte allo svolgimento di un attività, il cui scopo è perseguire gli obiettivi di un organizzazione.

Le risorse di un **sistema organizzativo** sono di matrice diversa, ad esempio, *persone*, *denaro* e *materiali* sono definibili come **risorse**.

Al di sotto del sistema organizzativo si trova il ***sistema informativo***, il quale rappresenta una componente del sistema organizzativo, il cui scopo è:

- *Acquisizione.*
- *Elaborazione.*
- *Conservazione.*

di dati, inoltre, il sistema informativo deve essere in grado di compiere delle **elaborazioni** producendo risultati di interesse (ad esempio, l'estrazione di un particolare dato). Il funzionamento di un ***sistema informativo*** può essere sintetizzato nel seguente schema:

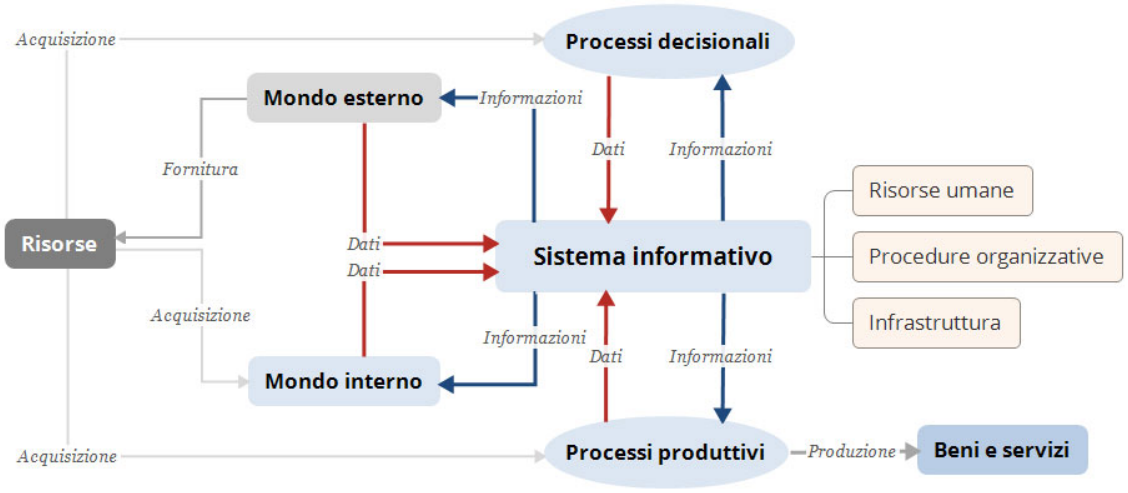


Figure 1: Organizzazione di un sistema informativo

L'implementazione automatizzata del sistema informativo viene anche definita come *sistema informatico.*

Dati e Informazioni

Un sistema informativo sfrutta, per poter funzionare, le *informazioni*, entità definite come

> *notizia, dato o elemento che consente di avere conoscenza più o meno esatta di fatti, situazioni o modi di essere*

Le informazioni sono generate dall'elaborazione e dalla caratterizzazione di un'altra entità sottostante che, prende il nome di *dato.* Secondo la definizione più generale un dato è definibile come

> *ciò che è immediatamente disponibile alla conoscenza, prima di ogni elaborazione.*

Un dato è quindi interpretabile nella sua *misura quantitativa*, ad esempio, la misurazione di un sensore di luce, la quale senza una caratterizzazione è un numero senza alcuna utiltà. Una volta caratterizzata correttamente il dato diviene informazione e diventa quindi utile al sistema informativo.

Basi di dati

Una ***base di dati*** è una *collezione di dati* utilizzata per rappresentare le informazioni di interesse in un sistema informativo.

Una base di dati viene però gestita da un software, il quale prende il nome di ***Data Base Management System***, il quale deve essere in grado di gestire collezioni di dati *condivise*, *grandi* e *persistenti*, garantendone in ogni caso **affidabilità** e **sicurezza** in maniera *efficace* e *efficente*.

Un **DBSM** è quindi un software che si occupa di interfacciarsi come tramite tra l'utente e la base di dati.

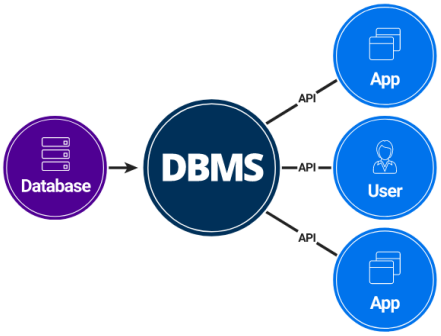


Figure 2: Schema di funzionamento di un DBMS

Ogni singola caratteristica della **base di dati** e del **DBSM** può essere analizzata singolarmente

- Una *base di dati* è **grande**: difatto una *base di dati* può anche avere delle dimensioni considerevoli. Di conseguenza i **DBSM** devono prevedere una gestione dei dati in memoria che permetta di elaborare quantitativi di dati coerenti con la grandezza della *base di dati*.
- Una *base di dati* deve essere **condivisa**, questa caratteristica è fondamentale per evitare la *ridondanza informativa* e l'*incoerenza* dei dati che si potrebbe avere utilizzando *file* privati diversi per ogni applicativo dell'azienda.

  Il problema che si viene a creare con la condivisione è la gestione degli accessi, difatti si deve evitare che due utenti possano apportare modifiche alla *base di dati* contemporaneamente, in modo da evitare uno stato di **incoerenza** dei dati. Per risolvere questa problematica è stato definito, insieme al DBSM, un **controllo di concorrenza**.
- Una *base di dati* è **persistente**, la sua durata di vita non si limita dunque alla sola esecuzione del programma, ma persiste per un tempo di vita indefinito.
- Un *DBSM* deve garantire **affidabilità**, questa caratteristica è fondamentale, essa permette che i dati si mantengano sempre in uno stato consistente, indipendentemente dal verificarsi o meno di guasti. Per garantire questa caratteristica sono stati implementati insieme al DBMS dei meccanismi di **salvataggio** e **ripristino**.
- Un *DBSM* deve garantire **privatezza**, tramite controlli sui permessi è fondamentale regolare la possibilità di accesso agli utenti che accedono al Database, limitando la loro possibilità di accesso alle sole aree necessarie allo svolgimento della particolare funzione aziendale.
- Un *database* deve essere **efficiente**, deve essere infatti in grado di compiere elaborazioni corrette in un lasso di tempo accettabile per l'utente.

Modello dati

Un *Modello Dati* è un insieme di concetti utilizzati per organizzare i dati di interesse e descriverne la struttura in modo che essa risulti comprensibile ad un elaboratore. Ogni modello dati fornisce *meccanismi di strutturazione*, del tutto analoghi ai costruttori dei vari linguaggi di programmazione.

Il *modello relazionale* dei dati più diffuso prende il nome di *Modello relazionale*, esso permette di definire tipi per mezzo del costruttore *relazione*, che consente di rappresentare i dati sotto forma di **record** a struttura fissa. Una relazione viene spesso rappresentata come una tabella

Nome	**Cognome**	**Anno**
Marco	Brambilla	1985
Giulio	Rossi	2004

Table 1: Tabella relazionale

Il modello relazionale è tutt'oggi il più diffuso, nonostante esistano anche altri modelli dati:

- **Modello gerarchico**: un modello definito agli albori della nascita dei **DBSM**, il quale sfrutta uno schema ad *albero*, quindi *gerarchico*.
- **Modello reticolare**: modello derivato dal *modello gerarchico*, sfrutta uno schema a grafo.
- **Modello a oggetti**: un modello nato come evoluzione del *modello relazionale*, il quale ha come scopo l'introduzione dei paradigmi della **OOP** sui **DBSM**.
- **Modello XML**: modernamento del *modello gerarchico*, a differenza del padre i dati non devono più sottostare ad uno schema rigido.
- **Modello semi-strutturato**: modello nato per superare le limitazioni del *modello relazionale*.

Schemi e modelli del database

Nelle basi di dati basate sul modello relazionale ogni relazione è costituita da due parti

- Una parte **invariante** nel tempo, la quale rappresenta l'intestazione della relazione, composta dal nome stesso di quest'ultima e dalle sue colonne (dette *attributi*).

Nome	**Cognome**	**Anno**

- Una parte **variabile** nel tempo, chiamata istanza della relazione, costituita dall'insieme, variante nel tempo, delle sue righe.

Marco	Brambilla	1985
Giulio	Rossi	2004

La nozione di modello descritta in precedenza può essere ulteriormente sviluppata su un architettura standard basata su tre livelli

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

L'architettura a livelli garantisce l'*indipendenza dei dati*, questa proprietà permette a utente e programmi di utilizzare il database ad un alto livello di astrazione. L'indipendenza dei database può essere a sua volta decomposta in due casi specifici:

- ***Indipendenza fisica***: consente l'interazione con il DBMS in modo indipendente dalla struttura fisica dei dati, ciò rende possibile modificare strutture fisiche senza influenza i programmi che interagiscono con il database.

- ***Indipendenza logica***: permette l'interazione con il livello esterno della base di dati indipendentemente dal livello logico. Ad esempio, aggiungendo uno schema esterno in base alle esigenze dell'utente senza però dover necessariamente modificare il livello logico.

è importante notare che all'interno di una base di dati tutte le interazioni avvengono tramite DBMS, che si pone come intermediario in ogni transazione tra i dati e l'utente.

Linguaggi per basi di dati

Tutte le operazioni fatte su una base di dati devono essere codificate da un linguaggio, questo linguaggio permette di compiere operazioni di vario tipo, dalle modifiche all'estrazione di dati. I linguaggi per basi di dati si distinguono in due categorie:

- **Data Definition Language** (*DDL*): utilizzato per definire schemi logici, fisici, esterni e le autorizzazioni di accesso.
- **Data Manipulation Language** (*DML*): utilizzato per la manipolazione delle istanza della base di dati.

Part I

Teoria delle basi di dati

Chapter 1

Il modello relazionale

Il modello relazionele si basa su due concetti, *relazione* e *tabella*. La nozione di relazione è strettamente correllata a tutta quella branca della matematica chiamata **insiemistica**, una branca che mette in relazione tutti i vari insiemi, i quali, invece, sono associabili al concetto di tabella.

Il modello gerarchico rispetta il concetto di *indipendenza dei dati*, questo modello prevede infatti la distinzione tra:

- livello fisico.
- livello logico.

gli utenti che accedono ai dati e i programmatori che sviluppano le applicazioni fanno riferimento al solo **livello logico**; Per accedere ai dati non è necessario conoscere la struttura fisica sottostante al livello logico.

Il concetto di relazione presenta anche esso delle diverse accezioni che cambiano in base a come cambia il loro contesto di utilizzo:

- *relazione matematica*: definita in ambito matematico, in particolare nella branca dell'**insiemistica**.
- *relazione*: definita secondo la definizione del modello relazionale.
- *relazione* o *partnership*: costrutto del modello entità-relazione che permette di descrivere legami tra entità del mondo reale.

Il modello relazionale impone uno schema rigido sulla rappresentazione dei dati.

1 Prodotto cartesiano

Nell'ambito dell'**insiemistica** una delle nozioni principali è quella di **prodotto cartesiano**. Assumiamo di avere due insiemi D_1 e D_2 (definiti anche come *domini della relazione*), si definisce come **prodotto cartesiano**:

$$D_1 \times D_2 \tag{1}$$

il sotto-insieme delle coppie (v_1, v_2) tali per cui $v_1 \in D_1$ e $v_2 \in D_2$.

Nell'ambito delle basi di dati, le *relazioni matematiche* vengono considerate come finite, in quanto una base di dati risiede su un dispositivo fisico di *dimensione finita*.

Il concetto di **prodotto cartesiano** può essere espresso anche su un numero insiemi indefinito, il concetto può essere infatti generalizzato nel seguente modo, dato $n > 0$ tale per cui $D_1, \ldots, D_n$ sono insiemi, *non necessariamente distinti*, si definisce come prodotto cartesiano di $D_1, \ldots, D_n$, indicato come $D_1 \times D_2 \times \cdots \times D_n$ l'insieme delle *n-tuple* ordinate $(v_1, v_2, \ldots, v_n)$ tali per cui $v_1 \in D_1, \ldots, v_n \in D_n$.

Il numero delle n componenti del prodotto cartesiano viene definito come *grado* della relazione, invece il numero di elementi della relazione viene definito come *cardinalità*.

All'intero di una base di dati tutte le relazioni sono definite come sottoinsiemi del **prodotto cartesiano**, ad esempio, una relazione che descrive le partite di calcio di un campionato, definita

Squadra casa	Squadra ospite	Goal casa	Goal ospiti
Milan	Inter	3	3
Juventus	Napoli	5	1

Table 1.1: Tabella relazionale

nel seguente modo
non è altro che un sottoinsieme del prodotto cartesiani con riferimenti all'insieme delle *stringhe* e degli *interi*:

$$\textbf{Stringa} \times \textbf{Stringa} \times \textbf{Intero} \times \textbf{Intero}$$

2 Relazioni con attributi

In base a quanto osservato fino ad ora, si definisce come *relazione matematica* **un'insieme di *n-tuple* ordinate** $(v_1, v_2, \ldots, v_n)$ tali per cui $v_1 \in D_1, v_2 \in D_2, \ldots, v_n \in D_n$.

All'interno delle basi di dati ogni *n-tupla* contiene dati tra loro *collegati*. facendo riferimento alla relazione

Squadra casa	Squadra ospite	Goal casa	Goal ospiti
Milan	Inter	3	3
Juventus	Napoli	5	1

Table 1.2: Tabella relazionale

posso asserire che i dati all'interno della *n-tupla* (Milan, Inter, 3, 3) sono tra loro legati, poiché se presi insieme definiscono una relazione ben specifica: La squadra di casa *inter* ha giocato contro la squadra ospite *Milan* totalizzando un risultato di *3* a *3*. È importante notare che le relazioni sono insiemi, di conseguenza:

- Non è definito un ordine tra le *n-tuple* della tabella, difatti due tabelle con le stesse *n-tuple* in ordine diverso rappresentano la stessa relazione.
- Le *n-tuple* di una relazione sono tra loro diverse, non vi possono essere dei duplicati. Il concetto può essere generalizzato come segue:

  Una tabella rappresenta una relazione se e solo se le sue n-tuple sono diverse.

All'interno dei vari domini di una relazione è altrsì definito un ordinamento, in quanto l'*i-esimo* valore di una qualsiasi *n-tupla* della relazione appartiene all'*i-esimo* dominio, ad esempio, se andassimo a scambiare i goal in casa con i goal della squadra ospite tutta la relazione cambierebbe di significato, in quanto verrebbero invertiti i domini.

Ci si rende facilmente conto che il modo in cui vengono rappresentati i dati all'interno delle basi di dati sia naturalmente simile a una struttura a *record*: una relazione non è altro che un'insieme di record omogenei, definiti quindi sugli stessi campi. Inoltre, per eliminare la necessità di un ordinamento tra i domini è sufficente assegnare un nome ad ogni campo, che viene rappresentato dall'intestazione della tabella, questo nome rappresenta quindi il compito svolto dal campo specifico, nel caso della relazione precedente: **Squadra casa**, **Squadra ospite**, *Goal casa*, **Goal opsiti**.

Per formalizzare la relazioni con attributi si procede definendo come D l'insieme di tutti i domini, si definisce come X l'un'insieme di attributi e si definisce come

$$dom : X \to D \tag{2}$$

una funzione che associa ad ogni attributo $x \in X$ un dominio $dom(x) \in D$. Inoltre, si definisce come n-upla su un insieme di attributi X una funzione che associa a ciascun attributo $a \in X$ un elemento, o valore, nel dominio di A ($dom(A)$).

Grazie a queste due definizioni siamo in grado di ricavare una nuova definizione di *relazione*:

Una relazione su X è un insieme di n-tuple su X

rispetto alla precedente relazione, utilizzando gli attributi, si può escludere la necessità di una tecnica di rappresentazione posizionale.

Se t è una tupla su X e x è un attributo di X allora $t[x]$ equivale al valore di t sull'attributo x

t[SquadraOspite] = Lazio

significa che l'attributo SquadraOspite $\in X$ appartenente al dominio delle stringhe e definito sulla tupla t ha valore *Lazio*. D'altro canto il simbolo t[Y] con $Y \subseteq X$, denota il valore della tupla sul sottoinsieme di attributi Y.

Si può ora procedere a riassumere tutte le definzioni relative al modello relazionale:

- Uno *schema relazionale* è costituito da un simbolo R definito come *nome della relazione* e un'insieme di attributi $X = \{A_1, A_2, \ldots, A_n\}$, il tutto indicato di solito come $R(X)$.
- Uno *schema di base di dati* indicato solitamente come R e definito come un'insieme di *schemi di relazione* $R = \{R_1(X_1), R_2(X_2), \ldots, R_n(X_n)\}$
- Un'*istanza di relazione* su uno schema R(X) è un'insieme r di n-uple sull'insieme di attributi X. Talvolta si utilizza la notazione $r(X)$ per indicare un'*istanza di relazione* sull'insieme di attributi X.
- Un'*istanza della base di dati* su uno *schema di basi di dati* è un'insieme di relazioni $r = \{r_1, r_2, \ldots, r_n\}$ dove ogni $1 \leq i \leq n$ è una relazione sullo schema $R_i(X)$

3 Vincoli di integrità

In una base di dati è opportuno evitare situazioni di **incoerenza dei dati**, a tale scopo è stato introdotto il meccanismo dei **vincoli di integrità**. Un **vincolo di integrità** può essere visto come un **predicato logico**, se il suo valore equivale a **vero** il predicato è soddisfatto, se il suo valore equivale a falso il predicato non è soddisfatto.

In generale, a uno schema di basi di dati associamo un insieme di **vincoli** e consideriamo **corrette** le istanze che soddisfano tutti i vincoli. È possibile classificare i vincoli a seconda degli elementi di una base di dati che ne sono coinvolti. Si possono distinguere in particolare due categorie di vincoli

- **Vincolo *intrarelazionale***: un vincolo si definisce come ***intrarelazionale*** se il suo soddisfacimento è definito rispetto a **singole relazioni della base di dati**. Questa classe di vincoli presenta un'ulteriore divisione:
  - **Vincolo di *tupla***: se il suo soddisfacimento può essere valutato su ciascuna *n-upla*, indipendentemente dalle altre.
  - **Vincolo di *dominio***: vincolo di *n-upla* che coinvolge un solo **attrbuto**.
- **Vincolo *interrelazionale***: un vincolo si definisce come ***interrelazionale*** se, il suo soddisfacimento, coinvolge più **relazioni**.

4 Vincolo di integrità referenziale

Un vincolo di integrità referenziale fra un'insieme di attributi X di una relazione R_1 e un'altra relazione R_2 è soddisfatto se i valori su X di ciascuna tupla dell'istanza di R_1 compaiono anche come valori della chiave **primaria** di R_2.

La definizione richiede di fare particolare attenzione alla composizione della chiave e il numero di attributi da cui essa è composta. Nel caso in cui la chiave di R_2 sia composta da un solo attributo B, consequenzialmente si avrà l'insieme X sarà composto da un solo attributo A, allora il vincolo di integrità referenziale fra l'attributo A di R_1 e l'attributo B di R_2 è soddisfatta se per ogni tupla

$t_1 \in R_1$ in cui l'attributo A **IS NOT NULL**, esiste una tupla $t_2 \in R_2$ per cui vale la seguente relazione

$$t_1[A] = t_2[B] \tag{3}$$

Nel caso più generale, dobbiamo fare attenzione al fatto che ciascuno degli attributi in X deve corrispondere a un preciso attributo della chiave primaria K di R_2, per garantire ciò è necessario definire un ordinamento in ambedue gli insiemi,

$$\begin{aligned} X &= A_1, A_2, \ldots, A_n \\ K &= B_1, B_2, \ldots, B_n \end{aligned} \tag{4}$$

per fare in modo che il **vincolo di integrità** sia rispettato è necessario che la seguente condizione venga rispettata

$$\forall i \in [1, n] \quad t_1[A_i] = t_2[B_i] \tag{5}$$

5 Chiavi e Superchiavi

Una ***chiave*** in linea di massima può essere vista come un insieme di attributi utilizzato per identificare univocamente le tuple di una relazione. La definizione può essere **formalizzata** si procede con due passi:

- Un insieme K di attributi è *super-chiave* di una relazione r se r non contiene due *n-uple* distinte t_1, t_2 tali per cui $t_1[K] = t_2[K]$.
- K è ***chiave*** di r se è una **super-chiave minimale** di r
  - una ***super-chiave minimale*** è una **super-chiave** per cui non esiste un'altra **super-chiave** $Q \subseteq K$.

Un osservazione importante è che essendo in ogni **relazione** le *n-uple* diverse tra di loro, si può dire che ogni **relazione** ha come **super-chiave** l'**insieme degli attributi su cui è definita**. Inoltre, l'esistenza delle chiavi garantisce l'accessibilità a tutti i valori della base di dati e la loro univoca identificabilità.

Uno dei problemi persistenti dell'SQL è il rischio della proliferazione dei valori nulli, ciò rende necessario porre dei limiti all'utilizzo di questi valori all'intero delle chiavi per mantenere la coerenza all'interno della nostra base di dati. Per evitare questo problema è sufficente porre una condizione, l'impossibilità di assegnare un valore nullo ad una particolare attributo chiave, attributo che prende il nome di ***chiave primaria***

Le chiavi sono uno strumento fondamentale per mettere in relazione relazioni diverse. Le chiavi definite su schemi di relazioni sono un tipo di **vincolo di integrità**, detto **vincolo di chiave**.

6 Operazioni di modifica

Esistono alcune operazioni di aggiornamento permettono di modificare o aggiornare i record presenti sul database. Operazioni di questo tipo possono però portare al mancato soddisfacimento di un determinato vincolo, a questo scopo il DBMS ha implementato una gestione per queste particolari situazioni

- Operazione di inserimento: viene inserito una nuova *n-upla* in una relazione, un operazione di questo tipo può portare alla violazione di
  - **vincoli di dominio**.
  - **vincoli di tupla**.

  per evitare problemi di questo tipo, il DBMS ha imposto che ogni qualvolta viene provato un inserimento scorretto, esso deve essere ignorato e non eseguito.
- **Operazione di cancellazione**: viene eliminato una *n-upla* presente in una relazione, un operazione di questo tipo può portare alla violazione di

– vincoli interrelazionali.

generalmente il DBMS, come anche nel caso precedente, impone la non possibilità di eseguire cancellazioni "scorrette", ma oltre a questa possibilità esistono altre due opzioni:

– **Eliminazione a cascata**: all'eliminazione di un record, si elimina a "cascata" anche tutti quei record per cui il vincolo di integrità referenziale andrebbe violato se eliminassi la *n-upla* inziale.
– **Inserimento dei valori NULL**.

Chapter 2

Algebra relazionale

Le interrogazioni sulle basi di dati sono definite come *operazioni di lettura* che, possono o meno, richiedere l'accesso a una o più tabelle. Per specificare il significato di una interrogazione è necessario specificare due **formalismi**:

- **Modo dichiarativo**: si specificano le proprietà del risultato.
- **Modo procedurale**: si specificano le modalità di generazione del risultato.

Per definire il **comportamento** delle interrogazioni in modo procedurale si utilizzano le espressioni dell'algebra relazionale, definendo il **risultato** dell'interrogazione in **modo dichiarativo** utilizzando le espressioni di calcolo relazionale.

Una **transazione** rappresenta un'insieme di operazioni da considerare come **indivisibili** e **corrette** anche in presenza di *concorrenza* e con *effetti definitivi*. Il contesto in cui nascono le transazioni è quello creato dal problema della concorrenza, si prenda come esempio la prenotazione del biglietto di un treno, *cosa succederebbe se due utenti diversi andassero a prenotare lo stesso posto allo stesso tempo ?* La base di dati deve gestire questo problema di concorrenza concedendo l'utilizzo in **mutua esclusione** del servizio, per un determinato lasso di tempo ad uno dei due utenti della base di dati.

Inoltre, nei DBMS esiste una porzione della base di dati che contiene una descrizione centralizzata dei dati, che può essere utilizzata dai vari programmi.

L'algebra relazionaled definisce il comportamento delle interrogazione, definendo le caratteristiche che il risultato deve avere utilizzando le espressioni dell'**alegbra relazionale**. Questo tipo di **algebra** definisce alcuni **operatori** fondamentali, i quali si possono suddividere in due macrocategorie

- Operatori su ***insiemi***.
- Operatori di ***relazione***.

1 Operazioni su insiemi

Esistono tre operatori fondamentali che eseguono manipolazioni su insiemi di elementi

- **Unione**: l'unione di due relazioni sullo stesso insieme di attributi X è una relazione su X che contiene le n-uple sia dell'una che dell'altra relazione.
- **Intersezione** L'intersezione di due relazioni sullo stesso insieme di attributi X è una relazione su X che contiene le *n-tuple* appartenenti a entrambe le relazioni.
- **Differenza**: La differenze tra due relazioni sullo stesso insieme di attributi X è una relazione su X che contiene le n-uple appartenenti alla prima relazione ma, che non appartengono alla seconda relazione.

2 Operatore di ridenominazione

Sia r una relazione su un insieme di attributi X e sia Y un altro insieme di attributi con la stessa **cardinalità**. Siano inoltre $A_1, A_2, \ldots, A_n$ e $B_1, B_2, \ldots, B_n$ rispettivamente un ordinamento per gli attributi in X e un ordinamento per quelli in Y. Allora la ridenominazione

$$\rho_{(B_1,\ldots,B_n \leftarrow A_1,\ldots,A_n)} \tag{1}$$

contiene una *n-upla* t' per ciascuna tupla $t \in r$, definita nel seguente modo: t' è una n-upla su Y e $t'[B_i] = t[A_i]$, per $i \in [1, k]$. L'operatore di ridenominazione cambia soltanto i nomi degli attributi senza andare in alcun modo a modificare quelli che sono i valori associati.

3 Operatore di selezione

[DA TERMINARE SUL LIBRO CON DEFINIZIONE FORMALE] Sia R una relazione su un insieme di attributi X e sinao $x_1, \ldots, x_n \in X$ attributi di R. La selezione sulla relazione R è definita come

$$\sigma_F(R) \tag{2}$$

dove F è una espressione booleana ottenuta come composizione di vari operatori logici e condizioni atomiche, le quali hanno forma

- $A\ \phi\ B$ dove ϕ è un operatore logico e A e B sono attributi di X.
- $A\ \phi\ k$ dove ϕ è un operatore logico e A è attributo di X e k una qualsiasi costante.

La condizione atomica è vera solo per valori non nulli in qualsiasi attributo.

3.1 Esempio sulla selezione

Si assuma di avere una relazione Impiegati composta nel seguente modo poniamo la seguente

Matricola	Cognome	Filiale	Stipendio
001	Rossi	Filiale A	\$3000
002	Bianchi	Filiale B	\$3500
003	Verdi	Filiale A	\$3200
004	Russo	Filiale C	\$3800
005	Romano	Filiale B	\$3300

Table 2.1: Esempio di tabella con attributi Matricola, Cognome, Filiale e Stipendio.

richiesta, *impiegati che guadagnano più di 3500 euro e che vivono a Milano.* Utilizzando l'operatore precedentemente definto

$$\sigma_{\text{Stipendio } >500 \text{ AND Citta = 'Milano'}}(\text{Impiegati}) \tag{3}$$

4 Operatore di proiezione

Data una relazione R(x) e un insieme di attributi $Y \subseteq X$, si definisce come **proiezione di Y su R** l'operazione

$$\pi_y(R) \tag{4}$$

Il risultato è una relazione su Y che contiene l'insieme delle *n-uple* di R ristrette ai soli attributi di Y. Ovviamente, essendo il risultato stesso un insieme, esso non può contenere duplicati.

Una proiezione può contenere al più tante *n-uple* quante ne ha l'operando, in particolare, se X è una super-chiave allora $\pi_X(R)$ contiene tante tuple quante ne ha R.

Matricola	Cognome	Filiale	Stipendio
001	Rossi	Filiale A	\$3000
002	Bianchi	Filiale B	\$3500
003	Verdi	Filiale A	\$3200
004	Russo	Filiale C	\$3800
005	Romano	Filiale B	\$3300

Table 2.2: Esempio di tabella con attributi Matricola, Cognome, Filiale e Stipendio.

4.1 Esempio sulla proiezione

Si assuma di avere una relazione Impiegati composta nel seguente modo poniamo la seguente richiesta, *impiegati che guadagnano più di 3500 euro e che vivono a Milano.* Utilizzando l'operatore precedentemente definto

$$\pi_{\text{Matricola, Cognome}}(\sigma_{\text{Stipendio>500 AND Citta = 'Milano'}}(\text{Impiegati})) \tag{5}$$

5 Operatore di Join

Il *join naturale* è un operatore che correla dati di relazioni diverse sulla base di attributi con lo stesso nome. Il risultato del join è costituito da una relazione sull'unione degli insiemi di attributi degli operandi e le sue tuple sono ottenute combinando le tuple degli operandi con valori uguali sugli attributi comuni. In generale il join naturale $R_1(X_1) \bowtie R_2(X_2)$ è una relazione definita su $X_1 \cup X_2$

$$R_1(X_1) \bowtie R_2(X_2) = \{\forall t | \exists t_1 \in R_1 \wedge t_2 \in R_2 \mapsto t[X_1] = t_1 \wedge t_2[X_2] = t_2\} = R_1(X_1) \cup R_2(X_2) \tag{6}$$

Se indichiamo come $X_C = X_1 \cap X_2$ l'insieme degli attributi comuni le due condizioni $t[X_1] = t_1$ e $t[X_2] = t_2$ implicano anche che $t[X_C] = t_1[X_C]$ e $t[X_C] = t_2[X_C]$, quindi che $t_1[X_C] = t_2[X_C]$. In generale la cardinalità del join è sempre compresa tra 0 e il prodotto delle cardinalità delle due relazioni

$$0 \leq R_1 \bowtie R_2 \leq R_1 \times R_2 \tag{7}$$

esistono tuttavia dei casi in cui si riesce a ridurre ancora di più questo intervallo

- Se $X_1 \cap X_2$ coinvolge una chiave di R_2 allora il numero di tuple è compreso tra 0 e R_1.
- Se $X_1 \cap X_2$ coincide con una chiave di R_2 e ivi è presente un vincolo di integrità referenziale rispetto a un attributo di R_1 allora il numero di elementi è equivalente a R_1.

Analizzando alcune proprietà del **join naturale** si osserva che

- è **commutativo**, quindi $R_2(X_2) \bowtie R_1(X_1) = R_1(X_1) \bowtie R_2(X_2)$.
- è **associativo**, quindi $(R_1(X_1) \bowtie R_1(X_1)) \bowtie R_3(X_3) = R_1(X_1) \bowtie (R_1(X_1) \bowtie R_3(X_3))$.

Pertanto, dove necessario, si potrebbe scrivere sequenze di join

$$\bowtie_{i=1}^{n} r_i \tag{8}$$

è importante però indagare i domini che si stanno utilizzando per questo **join n-ario**

- Se $X_1 = X_2$ allora $R_1(X_1) \bowtie R_2(X_2) = R_1(X_1) \cap R_2(X_2)$.
- Nel caso in cui i due attributi siano **disgiunti** bisogna tener conto che, essendo sempre il **risultato definito su** $X_1 \cup X_2$ e ciasciuna tupla da due tuple la condizione di uguaglianza dei valori su attributi comuni si decompone in una condizione sempre verificata. In questo caso particolare si dice che il **join** diviene un **prodotto cartesiano**.

5.1 Operatore di semijoin

Un operatore derivato da quello di **join** è il **semijoin**, che restituisce le tuple di una relazione che partecipano al **join naturale** di tale relazione con un'altra, escludendo quindi gli attributi dell'altra relazione. Date due relazioni $R_1(X_1)$ e $R_2(X_2)$ si definisce il **semijoin** come

$$R_1 \ltimes R_2 = t \mid t \in R_1 \text{ed esiste } t_2 \in R_2 \text{ con } t[X_1 \cap X_2] = t_2[X_1 \cap X_2] \tag{9}$$

6 Operatore di theta join

Il **theta join** è un operatore composto da un **join naturale** e una **selezione**. Siano $R_1(X_1)$ e $R_2(X_2)$ due relazioni, si definisce come **theta-join** l'operatore

$$R(X_1) \bowtie_F R(X_2) \tag{10}$$

dove F è una **condizione atomica**. Nello specifico, quando la condizione F fa utilizzo di un operatore di ugualiglianza, il **theta-join** viene chiamato **equi-join**.

7 Join esterni

La tendenza dell'operatore join di tralasciare le tuple di una relazione **che non hanno controparte nell'altra** si può rivelare **pericolosa** in alcuni casi, in quanto potrebbe omettere dati potenzialmente utili. A questo scopo, in **algebra relazionale** vi è la possibilità di utilizzare l'operatore di ***join esterno***. Questo operatore, tramite valori **nulli**, include all'interno del risultato dell'operazione di ***join*** anche tutte quelle n-uple che verrebbero altrimenti scartate. Esistono tre versioni di questo join

- **join esterno sinistro**: mantiene tutte le n-uple **del primo operando**, estendendo con valori **nulli** se necessario.
- **join esterno destro**: mantiene tutte le n-uple **del secondo operando**, estendendo con valori **nulli** se necessario.
- **join esterno completo**: mantiene **tutte le n-uple**, estendendo con valori nulli se necessario.

8 Divisione tra due relazioni

L'operatore di divisione è un **operatore derivato**, chiamato nel seguente modo per via della sua possibile interpretazione come **operatore inverso del prodotto cartesiano**. Dati due insiemi di **attributi disgiunti** X_1 e X_2, una relazione $R(X_1 \cup X_2)$ e una relazione $R_2(X_2)$ la divisione $R \div R_2$ è una relazione su X_1 che contiene le *n-uple* ottenute come proiezione di *n-uple* di r che si combinano con tutte le n-uple di R_2.

$$R \div R_2 = \{t_1 \text{su} X_1 | \forall t_2 \in R_2 \ \ \exists t \in R \mid t[X_1] = t_1 \wedge t[X_2] = t_2\} \tag{11}$$

In termini semplici, l'operatore **divisione** restituisce tutte le n-uple di una relazione che si combinano a tutte le n-uple di un'altra.

Essendo la divisione un operatore **non essenziale** esso viene espresso come composizione di altri operatori

$$R \div R_2 = \pi_{X_1}(R) - \pi_{X_1}(\pi_{X_1}(R) \times R_2) - R) \tag{12}$$

Osservando singolarmente i passi si nota che

- $\pi_{X_1}(R) \times R_2$ indica il prodotto cartesiano tra l'attributo su cui si sta cercando di fare la divisione e la seconda relazione, in questo modo è possibile creare tutte coppie.

- $\pi_{X_1}(R) \times R_2) - R$ tolgo alle coppie appena create tutti gli elementi della relazione originale, la tabella che ne risulta contiene quindi **tutte le coppie non presenti nella relazione originale**.
- $\pi_{X_1}(R) - \pi_{X_1}(\pi_{X_1}(R) \times R_2) - R)$ sottrae alla relazione le *n-uple* che non hanno una correlazione con tutte le *n-uple* di R_2.

9 Equivalenza di espressione

Due espressioni si definiscono **equivalenti** se e solo se **producono lo stesso risultato** qualunque sia l'istanza attuale della base di dati. Sfruttando l'equivalenzia siamo in grado di trovare espressioni che, all'apparenza producono meno risultati, ma sono meno **costose**.

L'equivalenza delle espressioni algebriche risulta particolarmente importante dal punto di vista applicativo, difatti quando le interrogazioni vengono tradotte in algebra relazionale dal DBMS viene valutato il costo **in termini di dimensioni del risultato** e quando viene rilevata un espressione equivalente con costo minore viene adottata quella. Esiste un insieme di equivalenze interessanti

- **Atomizzazione delle selezioni**: una congiunzione di selezioni può essere sostiuita da una sequenza di selezioni atomiche

$$\sigma_{F_1 \wedge F_2}(E) = \sigma_{F_1}(\sigma_{F_2}(E)) \tag{13}$$

- **Idempotenza delle proiezioni**: una proiezione può essere trasformata in una sequenza di proiezioni che eliminano i vari attributi in varie fasi

$$\pi_X(E) = \pi_X(\pi_{XY}(E)) \tag{14}$$

- **Push selection down**

$$\sigma_F(E_1 \bowtie R_2) = R_1 \bowtie \sigma_F(R_2). \tag{15}$$

- **Push projection down**: Siano E_1 e E_2 due relazioni definite rispettavamente sugli insiemi X_1 e X_2, se $Y_2 \subseteq X_2$ e $(X_1 \cap X_2) \subseteq Y_2$ vale la seguente relazione

$$\pi_{X_1 Y_2}(E_1 \bowtie E_2) = E_1 \bowtie \pi_{Y_2}(E_2) \tag{16}$$

10 Ottimizzazione delle interrogazioni

Il DBMS per elaborare le query sfrutta una particolare componente, chiamata **query processor**. Quando una interrogazione viene passata a questo componente, essa viene tradotta in un **linguaggio algebrico relazionale** interno al sistema. Esistono in generale **più possibili traduzioni di una interrogazione SQL** nel linguaggio algebrico (*equivalenza delle espressioni*). Il compito dell'**ottimizzatore** delle interrogazioni (**query optimizer**) è scegliere il piano di esecuzione più efficiente

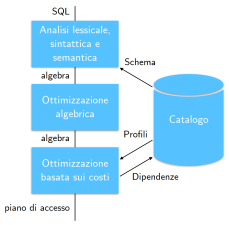


Figure 2.1: Elaborazione di una interrogazione

Per poter ottimizzare un'interrogazione è possibile osservarla anche sotto forma di albero, nel quale

- Le **foglie** sono rappresentate dai **dati**.
- I **nodi intermedi** sono rappresentati dagli **operatori**.

11 Viste

All'interno delle basi di dati risulta estremamente utile **fornire rappresentazioni diverse per gli stessi dati**. Nel modello relazionale, la tecnica destinata a questo scopo è quella delle **relazioni derivate**: *relazioni il cui contenuto viene generato a funzione di altre relazioni.* In una base di dati abbiamo la persistenza di due principali tipologie di relazioni:

- **Relazioni di base**.
- **Relazioni derivate**.

Esistono due tipologie di **relazioni derivate**:

- **Viste materializzate**: relazioni derivate memorizzate all'interno della base di dati.
- **Viste virtuali**: relazioni definite per mezzo di **funzioni**, ma non salvate esplicitamente nella base di dati.

Le **viste materializzate** hanno come svantaggio il **costo per rimanere memorizzate nella base di dati**. Inoltre, il loro svantaggio principale rimane l'**allineamento e la coerenza rispetto alle relazioni di base su cui esse sono state definite**. Le **viste virtuali**, invece, hanno come svantaggio principale di **dover essere ricalcolate ogni volta**, ma **non presentano il problema dell'allineamento dei dati**. Le viste vengono definite nei **sistemi relazionali** per mezzo di **espressioni** del **linguaggio di interrogazione**. Eventuali interrogazioni che si riferiscano alle viste **vengono risolte sostituendo alla vista la sua definizione**. In parole povere, **le viste sono delle tabelle che permettono di memorizzare relazioni utili ai fini di un'altra interrogazione.** Si assuma di avere uno schema di basi di dati R costituito dalle relazioni **Direzione** e **Impiegato**. Si supponga che il reparto delle risorse umane abbia bisogno sovente di

Afferenza

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B
Verdi	C

Direzione

Reparto	Capo
A	Mori
B	Bruni
C	Leoni

Figure 2.2: Schema Afferenza-Direzione

conoscere tutti gli impiegati che sono anche a loro volta dei capi. Per evitare di dover ripetere ogni volta l'interrogazione, si potrebbe procedere a memorizzarla in una vista:

$$\text{Supervisione} = \pi_{\text{Impiegato, Capo}}(\text{Afferenza} \bowtie \text{Direzione}) \tag{17}$$

12 Calcolo relazionale

Il termine **calcolo relazionale** fa riferimento a una famiglia di **linguaggi *dichiarativi* di interrogazione**, basati sul calcolo dei **predicati del primo ordine**. A differenza dell'**algebra relazionale** (che è un linguaggio **procedurale**), **il calcolo relazionale specifica soltanto le caratteristiche che dovrà avere il risultato**, senza andare a definire i passaggi per ottenerlo. Esistono molte versioni di **calcolo relazionale**; le due principali sono:

- Calcolo **relazionale** sui **domini**.
- Calcolo su **n-tuple** con dichiarazioni di **range**.

12.1 Calcolo relazionale sui domini

Le espressioni hanno la forma:

$$\{A_1 : x_1, \ldots, A_k : x_k \mid f\} \tag{18}$$

dove:

- $A_1 : x_1, \ldots, A_k : x_k$ è chiamata **target list**, il cui scopo principale è la **descrizione del risultato**. Si compone di due ulteriori componenti:
  - $A_1, \ldots, A_k$ sono **attributi distinti**.
  - $x_1, \ldots, x_k$ sono **variabili**.
- f è una formula (con **connettivi booleani**) definita secondo le seguenti regole:
  - Vi sono formule atomiche di due tipologie:
    - $R(A_1 : x_1, \ldots, A_k : x_k)$ dove $R(A_1, \ldots, A_k)$ è uno **schema di relazione** dove $x_1, \ldots, x_k$ sono **variabili**.
    - $x\theta y$ sono delle espressioni atomiche, dove θ indica un qualsiasi operatore di confronto.
  - Se f_1 e f_2 sono formule, allora lo sono anche $f_1 \wedge f_2$, $f_1 \vee f_2$ e $\neg f_1$.
  - Se f è una formula e x è una variabile, allora lo sono anche $\exists x(f)$ e $\forall x(f)$, dove $\exists$ e $\forall$ sono **quantificatori**. In particolare, questi **quantificatori** funzionano nel seguente modo:
    - $\exists x(f)$ è vera se esiste almeno un valore a che, sostituito alla variabile x, rende vera f.
    - $\forall x(f)$ è vera se per ogni possibile valore a per la variabile x, la formula f è vera.

Il risultato di questa espressione è una relazione su $A_1, \ldots, A_k$ che contiene **n-tuple** di valori $x_1, \ldots, x_k$ che rendono vera la formula f rispetto a un'istanza di base di dati a cui l'espressione è applicata.

Il calcolo sui **domini** presenta anche molti svantaggi, come la produzione di interrogazioni insensate o la necessità di importanti quantità di variabili. Questi sono solo alcuni dei motivi per cui è stata adottata un'altra forma di calcolo relazionale, basata su un concetto opposto: **l'indipendenza dal dominio**. L'indipendenza dal dominio è una caratteristica presente quando *il risultato di un'interrogazione non varia al variare del suo dominio di definizione*. Un **linguaggio si definisce *indipendente dal dominio* se e solo se tutte le sue espressioni lo sono**.

12.2 Calcolo sulle n-tuple

Le espressioni del calcolo sulle *n-tuple* hanno la seguente forma:

$$\{T \mid L \mid f\} \tag{19}$$

dove:

- T, nota anche come **target list**, denota la lista degli obiettivi dell'interrogazione. Questa può essere strutturata in vari modi:
  - $Y : x.Z$ se vogliamo solo gli attributi di X della variabile x che assumerà valori definiti in L e li chiameremo Y.
  - $x.Z \equiv Z : x.Z$ se vogliamo solo gli attributi Z della variabile x che assumerà valori definiti in L e non li rinomineremo.
  - $x* \equiv X : x.X$ se vogliamo tutti gli attributi della variabile x che assumerà valori in L.
- L è la **range list**, che contiene senza ripetizione tutte le variabili della target list, con la relazione associata da cui sono prelevati i valori assunti dalle variabili. L è quindi una dichiarazione di range, specifica l'insieme dei valori che possono essere assegnati alle variabili.
- f è la formula con:
  - atomi del tipo $x\theta y$.
  - connettivi logici.
  - quantificatori come nel calcolo sui domini.

Part II

Progettazione di una base di dati

Chapter 3

Metodologie e modelli per il progetto

1 Ciclo di vita di un sistema informativo

La progettazione di una base di dati costituisce solo una delle componenti del processo di sviluppo di un sistema informativo complesso; questa componente rientra in un complesso più grande, chiamato ***ciclo di vita*** del sistema informativo. Questo ***ciclo di vita*** comprende, generalmente, alcune fasi:

- **Studio di Fattibilità**: Serve a definire, in maniera precisa, i costi, le possibilità e le priorità di realizzazione di una **base di dati**.
- **Raccolta e Analisi dei requisiti**: Consiste nella raccolta e nell'individuazione di tutte le caratteristiche richieste per la base di dati; questa fase richiede l'interazione con gli utenti utilizzatori del sistema.
- **Progettazione**: Si divide generalmente in *progettazione dati* e *progettazione delle applicazioni*. Nella prima fase si individua la struttura e l'organizzazione che i dati dovranno avere, nella seconda si definiscono le caratteristiche dei programmi applicativi che sfrutteranno la base di dati.
- **Implementazione**: Viene prodotto il codice dei programmi e costruita la **base di dati**, il tutto seguendo le linee guida definite durante la fase di **progettazione**.
- **Valutazione e Collaudo**: Serve a verificare il corretto funzionamento e la qualità del sistema informativo.
- **Funzionamento**: In questa fase il **sistema informativo** diventa operativo ed esegue i compiti per i quali era stato inizialmente progettato.

Opzionalmente viene anche introdotta la fase di **prototipazione**, fase nella quale, mediante opportuni strumenti software, viene realizzato un **prototipo** del sistema informativo.

La progettazione di una **base di dati** richiede che venga fatta una trattazione sul significato del termine ***metodologia di progettazione***, termine ombrello che racchiude sotto di sé fasi specifiche:

- Una ***decomposizione*** dell'intera attività di progetto in passi successivi indipendenti tra di loro.
- Una serie di ***strategie*** da seguire nei vari passi e alcuni ***criteri*** per la possibile scelta di alternative (se necessario variare rispetto al progetto originale).
- Alcuni ***modelli di riferimento*** per descrivere i dati in ingresso e uscita dalle varie fasi.

In aggiunta, una ***metodologia di progettazione*** deve garantire:

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

La **metodologia di progettazione** che più si è fatta strada negli ultimi anni è articolata in tre fasi, separando nettamente le decisioni relative a *cosa* rappresentare in una base di dati e *come* rappresentarlo:

- **Progettazione concettuale**: Il suo scopo è quello di rappresentare le specifiche informali della realtà di interesse in termini di una descrizione **formale** e **completa**. Il prodotto di questa fase viene chiamato ***modello concettuale*** dei dati. Questo modello permette di rappresentare il **contenuto informativo** della base di dati, senza preoccuparsi né delle modalità con le quali queste informazioni verranno codificate in un sistema reale, né dell'efficienza dei programmi che faranno uso di queste informazioni.
- **Progettazione logica**: Consiste nella traduzione dello schema concettuale definito nella fase precedente, in termini del modello di rappresentazione dei dati adottato dal **DBMS** a disposizione. Il prodotto di questa fase viene chiamato ***schema logico*** della **base di dati** e fa riferimento a un **modello logico** dei dati. In questa fase vi è ancora una sostanziale indipendenza rispetto al modello fisico. Inoltre, sempre in questa fase vengono utilizzati dei criteri di ottimizzazione che, nel caso del **modello relazionale di dati**, prendono il nome di **normalizzazione**.
- **Progettazione fisica**: In questa fase lo schema logico viene completato con la specifica dei parametri fisici di memorizzazione dei dati. Il prodotto di questa fase prende il nome di **modello fisico**.

Occorre distinguere due caratteristiche durante la fase di progettazione:

- ***specifiche sulle operazioni***, che riguardano l'uso che utenti e applicazioni fanno della base di dati.
- ***specifiche sui dati***: che riguardano il contenuto della base di dati.

Questa distinzione deriva dalla fase in cui queste specifiche vengono prese in considerazione.

Il risultato finale della **progettazione** è composto sia dallo schema fisico, che dallo schema logico e da quello concettuale; ognuno di questi copre una specifica funzione, permettendo di avere una vista completa su tutto il progetto.

2 Modellazione concettuale

Nel paragrafo precedente è stata data, in generale, la *sequenza di fasi* e i *vincoli da seguire* per la **progettazione di una base di dati**. Da questa analisi è emerso che il prodotto finale di questa progettazione comprende **tre schemi**: lo schema **concettuale**, lo schema **logico** e lo schema **fisico**. In questo paragrafo ci concentreremo sul primo schema, quello **concettuale**

Il **modello E-R** (*Entity-Relationship*) è un **modello concettuale di basi di dati** che fornisce *strutture* e *costrutti* atti a **descrivere la realtà** di interesse in modo del tutto indipendente dallo schema di funzionamento del calcolatore. All'interno di questi diagrammi possiamo distinguere **tre elementi principali**:

- **Entità** (*entity*): Rappresentano gli oggetti che hanno **proprietà comuni** e un'**esistenza autonoma**. Rispetto al **modello relazionale**, un'occorrenza di un'entità non rappresenta dei valori che identificano l'oggetto, ma è l'oggetto stesso. Quindi, un'occorrenza **ha un'esistenza indipendente dalle proprietà ad essa associate**.

- **Relazione** (*relationship*): Rappresentano i **legami logici che esistono tra entità**. Un'**occorrenza di relazione** è una **n-upla costituita da occorrenze di entità**, una per ciascuna entità coinvolta. In uno **schema E-R**, ogni relazione ha un **nome** che la **identifica univocamente**.

  Inoltre, una relazione tra due entità è a tutti gli effetti una **relazione matematica** e di conseguenza un **sottoinsieme del prodotto cartesiano**. Questo fatto implica l'**impossibilità di esistenza di duplicati**.

  Infine, vi la possibilità di definire **relazioni n-arie**, quindi relazioni tra un numero **indefinito di enitità** e anche relazione **ricorsive**, quindi relazioni tra **un enitità e se stessa**.

- **Attriubuti**: Descrivono le **proprietà elementari di entità** o **relazioni** che sono di interesse ai fini dell'applicazione. All'interno di un diagramma, potrebbe essere utile raggruppare più attributi, considerandoli come se fossero un singolo attributo. Questi attributi prendono il nome di **attributi composti**.

Inoltre, oltre a tutti i costrutti fisici che sono stati elencati esistono anche altri costritutti, che sono invece più logici e concettuali

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

  Per esempio, si consideri una relazione tra **Poliziotto** e **Dipartimento** che prende il nome di entità. Sappiamo che a un **Dipartimento** possono essere associati n **poliziotti**, ma che un singolo **poliziotto** può essere assegnato solo a un **dipartimento**. Quindi n occorrenze di **Poliziotto** possono partecipare alla relazione con una singola occorrenza di **Dipartimento**. Questa tipologia di cardinalità prende il nome di ***one-to-many***.

  In realtà, non è necessario nemmeno che un'entità partecipi a una relazione. È infatti sufficiente specificare la relazione in questo modo $(0, n)$, per indicare una relazione "***zero-to-many***".

- **Cardinaltà degli attributi**: Possono essere specificate per gli attributi di **entità** o **relazioni** e **descrivono il numero massimo e minimo di valori dell'attributo associabili a una singola occorrenza**. Nella maggior parte dei casi, essendo questa cardinalità pari a (1,1), essa viene omessa.

- **Idenitificatori delle entità**: Vengono specificati per ogni entità e identificato l'insieme di attributi che permette di identificare univocamente le occorrenze dell'entità. Esistono due tipologie principali di identificatori:

  - ***Identificatore esterno***: se l'identificatore di un'entità deve essere costituito attraverso attributi di un'altra entità che partecipa a una relazione con l'entità di cui stiamo cercando di comporre l'identificatore. Una **identificazione esterna** è possibile solo in presenza di una cardinalità $(1, 1)$.
  - ***Identificatore interno***: se un insieme di attributi interno alla classe è in grado di comporre un identificatore univoco.

- **Generalizzazione**: rappresenta legami logici tra un'entità detta **entità padre** e un insieme di entità, dette **entità figlie** o **specializzazione**. In questo ambito valgono alcune proprietà:

  - ***Generalizzazione Totale-Parziale***: Una **generalizzazione** è **totale** se ogni occorrenza dell'entità genitore è anche occorrenza di almeno una delle entità figlie. Se invece possono esistere occorrenze dell'entità genitore che non sono occorrenze dell'entità figlie, la relazione si definisce **parziale**.
  - ***Generalizzazione Esclusiva-Non Esclusiva***: Una generalizzazione è esclusiva se ogni occorrenza dell'entità genitore compare al massimo in una singola occorrenza dell'entità figlia; altrimenti, se un'occorrenza dell'entità padre può apparire in più occorrenze dell'entità figlia, la generalizzazione si definisce **non esclusiva**.

2.1 Regole aziendali

Uno degli strumenti più usati dagli analisti di sistemi informativi per la descrizione di proprietà di un'applicazione che non si riesce a rappresentare direttamente con modelli concettuali è quello delle **regole aziendali**. Questa accezione deriva dal fatto che, nella maggior parte dei casi, quello che si vuole esprimere è proprio una regola del particolare dominio applicativo che stiamo considerando.

I progettisti, in generale, utilizzano il termine in modo più ampio per indicare una qualunque informazione che definisce o vincola qualche aspetto di un'applicazione. In particolare, le ***business-rule*** possono essere:

- la **descrizione di un concetto rilevante per l'applicazione**, ovvero la definizione precisa di un'entità, di un attributo o di una relazione del modello E-R. Ad esempio, potrebbe essere descritto in modo particolare il modo in cui il direttore vuole che i suoi dipendenti vengano descritti nella base di dati.
- un **vincolo di integrità** sui dati dell'applicazione stessa, come ad esempio, la cardinalità di una particolare relazione.
- una **derivazione**, ovvero, un concetto ottenibile mediante un'operazione matematica rispetto ad altri elementi dello schema.

La mancanza di uno standard rende enormemente complessa l'esplicazione di queste **business-rule**, ciò obbliga infatti i progettisti a dover sfruttare il linguaggio naturale, andando a strutturarlo in modo che la condizione non contenga la risoluzione al problema. Esso infatti è un problema puramente realizzativo e non deve essere considerato nella parte concettuale. Tutte queste caratteristiche convergono nell'uso delle **asserzioni**, condizioni le quali devono essere **sempre verificate nella base di dati**, strutturate nella forma

> <*concetto*> deve/non deve <*espressione sui concetti*>

Assumendo di avere una base di dati che rappresenta una scuola, potrebbero essere definite le seguenti asserzioni:

- *uno studente non deve essere in due classi diverse contemporaneamente.*
- *la somma dei voti di uno studente deve essere uguale al numero di verifiche svolte durante l'anno.*

Le regole aziendali riguardanti le derivazioni possono essere invece espresse come:

> <*concetto*> si ottiene <*operazione sui concetti*>

Riferendoci sempre alla base di dati scolastica si ha:

- *la media dei voti di uno studente si ottiene come valore totale dei voti diviso il numero di essi.*
- *il numero di assenze totali si ottiene calcolando le assenze dall'inizio dell'anno fino alla fine.*

2.2 Tecniche di documentazione

Uno schema E-R va correado da una documentazione si supporto che ne faciliti l'interpretazione e descriva le proprietà che non possono essere esplicate sullo schema. La documentazione dei vari concetti può essere prodotta facendo uso di un dizionario dei dati; Esso comprende due tabelle

Entità	**Attributi**	**Identificatore**	**Descrizione**
Studente	Matricola, Nome, Cognome, Data di nascita	Matricola	Entità rappresentante uno studente
Corso	Codice corso, Nome del corso, Crediti	Codice corso	Entità rappresentante un corso universitario
Docente	ID Docente, Nome, Cognome, Dipartimento	ID Docente	Entità rappresentante un docente

Relazione	**Entità Coinvolte**	**Cardinalità**	**Descrizione**
Frequenta	Studente, Corso	(0,n) - (1,n)	Uno studente frequenta n corsi

La prima tabella descrive le entità, con i loro attributi, identificatori, nome e una descrizione informale. La seconda tabella descrive invece le relazioni, speficiando una descrizione generica, le entità coinvolte con le rispettive cardinalità e eventuali attributi.

Chapter 4

Progettazione concettuale

1 Raccolta e analisi dei requisiti

Per *raccolta dei requisiti* si intende la completa individuazione dei problemi che l'applicazione da realizzare deve risolvere e le caratteristiche che tale applicazione dovrà avere. Per caratteristiche del sistema si intendono sia gli aspetti statici che gli aspetti dinamici. I requisiti vengono inizialmente raccolti in modo ambiguo e disordinato. Tramite *l'analisi dei requisiti* è possibile ordinare le caratteristiche, i requisiti e tutte le specifiche che il sistema dovrà avere. I requisiti di un'applicazione provengono spesso da fonti diverse:

- Gli **utenti dell'applicazione**. In questo caso, i dati sono ottenuti mediante delle interviste, anche ripetute, oppure attraverso una documentazione scritta che gli utenti possono aver predisposto.
- Tutta la *documentazione esistente*, che ha qualche attinenza con il problema.
- Eventuali *realizzazioni pre-esistenti*, magari sistemi informativi già in uso o altri applicativi.

Come abbiamo visto, l'interazione con l'utente è una fase fondamentale per la progettazione di un sistema informativo, nonostante possa presentare delle criticità, prima tra tutte, la relatività del modo di esprimersi. Ogni utente ha un vocabolario diverso e potrebbe intendere il medesimo concetto con termini diversi (professore, docente). È importante quindi cercare di definire un vocabolario nel quale siano considerati tutti i termini che potrebbero esprimere un concetto utile alla progettazione. Il problema descritto precedentemente ci fa rendere conto di quanto questa fase di progettazione sia poco standardizzabile. Nonostante ciò, si possono definire delle linee guida molto generali:

- **Scegliere il corretto livello di astrazione**: cercare di utilizzare termini né troppo specifici, né troppo astratti, in modo che siano di facile comprensione.
- **Standardizzare la struttura delle fasi**: Nella specifica di requisiti è preferibile utilizzare sempre lo stesso stile sintattico.
- **Evitare frasi contorte**: Le definizioni devono essere semplici e chiare.
- **Unificare i termini**.
- **Esplicitare il riferimento tra termini**.
- **Costruire un glossario dei termini**.

Dopo aver individuato le varie ambiguità e le imprecisioni, queste vanno eliminate sostituendo i termini non corretti con termini più adeguati

2 Pattern di progetto

I *pattern di progetto* (o *design patterns*) sono soluzioni generiche e riutilizzabili a problemi comuni che si presentano durante il processo di sviluppo software. Sono frutto dell'esperienza accumulata dalla comunità degli sviluppatori e rappresentano le migliori pratiche per risolvere problemi specifici di progettazione software. I pattern di progetto non sono modelli o pezzi di codice pronti all'uso, ma piuttosto descrizioni strutturate di come organizzare il codice in maniera efficace, mantenendo un alto grado di manutenibilità, estensibilità e leggibilità.

I pattern di progetto possono essere suddivisi in tre categorie principali:

- **Pattern Creazionali**: Questi pattern riguardano la modalità di creazione degli oggetti, fornendo delle soluzioni per gestire l'istanza di oggetti in maniera flessibile e indipendente dal contesto. Alcuni esempi noti sono:
  - **Singleton**: Garantisce che una classe abbia una sola istanza e fornisce un punto di accesso globale a quell'istanza.
  - **Factory Method**: Definisce un'interfaccia per creare un oggetto, ma lascia alle sottoclassi la decisione di quale classe istanziare. Questo pattern permette alle classi di delegare l'istanziazione ai sottotipi.
  - **Abstract Factory**: Fornisce un'interfaccia per creare famiglie di oggetti correlati o dipendenti senza specificare le loro classi concrete.
- **Pattern Strutturali**: Questi pattern si concentrano sulla composizione delle classi o degli oggetti per formare strutture più complesse e mantenibili. Gli esempi più comuni includono:
  - **Adapter**: Permette l'interazione tra due interfacce incompatibili tramite la creazione di una classe intermedia che "adatta" una delle interfacce all'altra.
  - **Decorator**: Fornisce un modo flessibile di aggiungere responsabilità agli oggetti dinamicamente, senza modificare le classi originali.
  - **Composite**: Permette di trattare oggetti individuali e gruppi di oggetti in maniera uniforme, creando strutture gerarchiche ad albero.
- **Pattern Comportamentali**: Questi pattern si occupano dell'interazione tra gli oggetti, definendo la comunicazione e l'assegnazione di responsabilità. Alcuni esempi noti includono:
  - **Observer**: Definisce una dipendenza uno-a-molti tra oggetti, in modo che quando uno stato cambia, tutti i suoi dipendenti vengono notificati e aggiornati automaticamente.
  - **Strategy**: Consente di definire una famiglia di algoritmi, incapsularli in classi separate e renderli intercambiabili tra loro. In questo modo, l'algoritmo utilizzato da un oggetto può variare a seconda delle necessità.
  - **Command**: Incapsula una richiesta come un oggetto, permettendo così di parametrizzare gli oggetti con le richieste, mettere in coda o eseguire richieste in momenti diversi.

L'adozione dei pattern di progetto permette di affrontare problemi di progettazione complessi con soluzioni già collaudate, migliorando la qualità del software. Inoltre, facilitano la comunicazione tra sviluppatori, poiché offrono un linguaggio comune per descrivere problemi e soluzioni ricorrenti. Tuttavia, è importante utilizzare i pattern con giudizio, evitando l'overengineering e scegliendo il pattern più appropriato per il contesto specifico.

3 Strategie di progettazione

Lo sviluppo di uno schema concettuale a partire dalle sue specifiche può fare uso di alcune strategie di progettazione, le principali sono:

- **Strategia Top-Down**: In questa strategia, lo schema concettuale viene prodotto mediante una serie di raffinamenti successivi, a partire da uno schema iniziale che descrive tutte le specifiche con pochi concetti astratti. Man mano che lo schema viene raffinato, aumenta conseguenzialmente la sua precisione e il suo dettaglio. Questa strategia può essere immaginata come una serie di piani: partendo dal più alto si scende, raffinando sempre di più il progetto. Nel passaggio di livello, lo schema viene modificato tramite trasformazioni elementari dette **primitive di trasformazioni top-down**. Il vantaggio di questa strategia è che il progettista può, almeno nelle fasi iniziali, descrivere le specifiche dei dati trascurandone i dettagli, salvo poi approfondire singolarmente le componenti del progetto. Questa strategia di progettazione funziona bene quando si possiede fin dall'inizio una visione globale del sistema, ma risulta difficile quando si lavora su sistemi di una certa complessità.
- **Strategia Bottom-Up**: In questa strategia, le specifiche iniziali sono suddivise in componenti sempre più piccole, fino a quando esse descrivono frammenti elementari del progetto. A questo punto, le varie componenti vengono rappresentate da singoli schemi concettuali, anche composti da entità singole. I vari schemi ottenuti vengono fusi fino a giungere, attraverso una completa integrazione di tutte le componenti, allo schema concettuale finale. Anche in questo caso, lo schema finale viene ottenuto attraverso delle trasformazioni **primitive di trasformazioni bottom-up** che introducono nello schema concetti non presenti precedentemente e in grado di descrivere aspetti della realtà di interesse che non erano stati rappresentati. Il vantaggio della strategia bottom-up è che si adatta ad una decomposizione del problema in componenti più semplici, facilmente individuabili, il cui progetto può essere affrontato anche da progettisti diversi. Lo svantaggio principale sta, invece, nelle operazioni di integrazioni che, su progetti di grande scala potrebbero risultare estremamente complesse.
- **Strategia Inside-Out**: In questa strategia si identificano i concetti importanti e successivamente, a macchia d'olio. Si rappresentano cioè prima i concetti in relazione con i concetti iniziali, per poi muoversi verso quelli più lontani attraverso una navigazione tra le specifiche. Il vantaggio è che la strategia non richiede nessuna integrazione, ma presenta comunque una grande complessità per quanto riguarda l'identificazione dei concetti importanti.

Combinando i vantaggi delle tre strategie descritte sopra si riesce a ottenere una strategia mista. Il progettista, oltre a dividere lo schema in tante piccole componenti, come nel **bottom-up**, ma allo stesso tempo definisce uno schema *scheletrico* generale, contenente i concetti principali dell'applicazione.

4 Qualità di uno schema concettuale

Quando si struttura un modello concettuale, è fondamentale che siano garantite alcune proprietà generali. Ognuna di queste proprietà ha un modo di essere verificata, esse si dividono in

- **Correttezza**: Uno schema può essere definito **corretto** quando *utilizza correttamente i costrutti messi a disposizione della convenzione sui diagrammi E-R*. Come nella normale programmazione, gli errori possono essere **semantici** o **sintattici**; i primi riguardano un uso non ammesso di determinati costrutti, i secondi invece riguardano un uso di costrutti senza però che sia rispettata la loro definizione.
- **Completezza**: Uno schema concettuale è **completo** quando rappresenta tutte le **caratteristiche** richieste e permette inoltre di operare tutte le **operazioni** richieste.
- **Leggibilità**: Uno schema concettuale è **leggibile** quando rappresenta i concetti in modo intuitivo e di facile comprensione. Allo scopo di rendere leggibile un modello concettuale, è necessario adottare delle convenzioni, ad esempio, usare uno standard definito per i nomi delle relazioni o delle entità, oppure ponendo gli elementi più importanti del modello al centro della griglia, ecc...
- **Minimalità**: Uno schema è **minimale** quando tutte le specifiche sui dati sono rappresentate una sola volta in tutto lo schema, quindi, quando **non esistono ridondanze**.

Chapter 5

Progettazione logica

1 Dal modello concettuale al modello logico

Una volta giunti al termine della progettazione concettuale, si passa a un nuovo capitolo della progettazione di una base di dati, la **progettazione logica**. Questa fase, però, non può essere svolta basandosi esclusivamente sullo schema concettuale precedentemente creato; esso va rielaborato. Il processo di rielaborazione del modello E-R prende il nome di **ristrutturazione**, un processo che ha come scopo l'ottimizzazione dello schema E-R prodotto.

Quando si ristruttura il diagramma E-R, si utilizzano due parametri:

- **Il modello concettuale** prodotto precedentemente
- Il **carico applicativo** previsto.

Il risultato di questa fase è uno schema che non rispetta più i criteri generali dello schema E-R, in quanto costituisce una rappresentazione dei dati che tiene maggiormente conto degli aspetti realizzativi.

1.1 Analisi delle prestazioni

L'analisi delle prestazioni di uno schema E-R è una fase fondamentale, che permette di attuare delle ottimizzazioni prima della fase di traduzione al modello logico. L'ottimizzazione avviene su **indici di prestazione**, i quali non sono direttamente collegati alle prestazioni generali della **base di dati**, poiché le prestazioni nel loro senso generale dipendono anche dalle caratteristiche fisiche della macchina che ospita la **base di dati**. Generalmente, vengono valutati due indici:

- **Costo di un'operazione**: viene valutato in termini di occorrenze e associazioni (o relazioni) che *mediamente vanno visitate per rispondere a un'operazione sulla base di dati.*
- **Occupazione di memoria**: viene valutata in termini di spazio di memoria necessario per memorizzare i dati descritti dallo schema.

Questa valutazione richiede di conoscere alcune informazioni riguardo allo schema:

- **Volume dei dati**: numero di occorrenze di ogni entità e associazione dello schema e dimensioni di ciascun attributo. Tutti questi dati vengono racchiusi nella **tabella dei volumi**.
- **Caratteristiche delle operazioni**: tipo dell'operazione, frequenza e dati coinvolti. Tutti questi dati vengono racchiusi nella *tabella delle operazioni.*

Successivamente, si procede definendo uno **schema di operazione**, che consiste nel frammento dello schema E-R interessato dall'operazione, sul quale viene disegnato il cammino logico da percorrere per accedere alle informazioni di interesse.

Inoltre, è importante definire anche una **tavola degli accessi**; avendo infatti a disposizione le informazioni ricavate precedentemente, è possibile fare una stima del costo di un'operazione sulla base di dati, *confrontando il numero di accessi alle occorrenze di entità e associazioni necessari a eseguire l'operazione.* Il meccanismo è basato su poche e semplici regole:

- Si conta il numero di accessi alle occorrenze del primo concetto presente nel cammino logico dello *schema di operazione*, supponendo che il recupero di un'occorrenza costi un singolo accesso.
- Si prosegue nel cammino assumendo che, avendo recuperato nei passi precedenti un'occorrenza, possiamo accedere tramite essa a tutte le occorrenze di altre entità o relazioni che coinvolgono l'occorrenza iniziale.

Negli appendici è possibile trovare qualche nota sul calcolo della **tavola degli accessi**.

1.2 Ristrutturazione dello schema E-R

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

- **Analisi delle ridondanze**
- **Eliminazione delle generalizzazioni**
- **Partizionamento o accorpamento di entità e associazioni**
- **Scelta degli identificatori principali**

Analisi delle ridondanze In generale, si può definire come **ridondanza** la presenza di un dato che può essere derivato. In un diagramma E-R esistono diversi tipi di ridondanze, le più frequenti sono:

- **Attributi derivabili da attributi della stessa entità**: sono attributi che potrebbero essere ricavati facilmente attraverso delle operazioni.
- **Attributi derivabili da attributi di altre entità**
- **Attributi derivabili da conteggio delle occorrenze**
- **Associazioni derivabili dalla composizione di altre associazioni**

La presenza di un attributo derivabile non è sempre da considerarsi un problema; se da una parte vi è una maggiore occupazione di memoria, dall'altra vi è una riduzione del numero di accessi necessari a calcolarlo.

Eliminazione delle generalizzazioni Vista la mancanza di un metodo per rappresentare direttamente le **generalizzazioni**, risulta spesso necessario trasformarle utilizzando altri costrutti del diagramma E-R. Per rappresentare le generalizzazioni con costrutti del modello E-R, abbiamo fondamentalmente tre possibilità:

- **Accorpamento nell'entità padre**: questo tipo di trasformazione risulta particolarmente utile quando si hanno generalizzazioni totali ed esclusive che non hanno associazioni con una specifica entità figlia.
- **Accorpamento del genitore nelle entità figlie**: l'entità genitore viene eliminata e i suoi attributi, identificatori e associazioni vengono inseriti nelle entità figlie. Questo tipo di trasformazione è necessaria quando la generalizzazione è totale, oppure quando ci sono associazioni relative solo alle specializzazioni dell'entità padre.
- **Sostituzione con associazioni**: la generalizzazione viene trasformata in due relazioni **uno-a-uno** che legano l'entità genitore con le entità figlie, non vi è quindi trasferimento di attributi o associazioni. Questo tipo di trasformazione risulta molto utile quando la generalizzazione non è **totale** e ci sono operazioni relative sia all'entità padre che alle entità figlie.

Quando si hanno generalizzazioni su più livelli, si può procedere analogamente, analizzando una generalizzazione per volta partendo dal fondo.

Partizionamento e accorpamento di concetti Entità e associazioni in uno schema E-R possono essere partizionati o accorpati per garantire una maggiore efficienza delle operazioni, basandosi sul seguente principio: gli accessi si riducono separando attributi di uno stesso concetto che vengono acceduti da operazioni diverse e raggruppando attributi di concetti diversi che vengono acceduti dalle stesse operazioni. Questa fase ha diverse operazioni possibili:

- **Partizionamenti di entità**: questa ristrutturazione è conveniente per operazioni che coinvolgono frequentemente l'entità originaria. A loro volta, i partizionamenti si dividono in due tipologie:
  - **Partizionamento verticale**: la suddivisione di un concetto operando sui suoi attributi.
  - **Partizionamento orizzontale**: la suddivisione avviene su occorrenze diverse della stessa entità.
- **Eliminazione degli attributi multivalore**: un particolare tipo di partizionamento riguarda l'eliminazione di attributi multivalore, i quali, come per le generalizzazioni, non hanno un modo per essere gestiti direttamente dal modello relazionale. Un attributo multiplo può essere trasformato in un'entità con un'associazione tra l'entità a cui era legato e se stesso.
- **Accorpamento di entità**: l'accorpamento di entità è l'operazione inversa al partizionamento. Questo genere di operazione viene attuato su associazioni **uno-a-uno**, le quali, nella maggior parte dei casi, sono accorpabili come unica entità.

Scelta degli identificatori principali La scelta degli identificatori principali è essenziale nella traduzione verso il modello relazionale, in quanto, in questo modello, le chiavi permettono di stabilire legami tra dati in relazioni diverse. La scelta deve avvenire in base a criteri specifici:

- Gli attributi con possibili valori **nulli** non possono essere presi in considerazione come identificatore principale.
- Un identificatore composto da uno o da pochi attributi è da preferire a identificatori costituiti da molti attributi.
- Un identificatore interno con pochi attributi è da preferire rispetto a un identificatore esterno.
- Un identificatore che viene utilizzato sovente dalle operazioni per l'accesso alle occorrenze di un'entità è da preferire rispetto agli altri.

Se nessuno degli identificatori candidati soddisfa i requisiti, è possibile introdurre un identificatore *codice*, nonostante questi identificatori debbano essere in qualche modo memorizzati, in quanto potrebbero risultare utili per delle operazioni (*identificatori secondari*).

1.3 Traduzione verso il modello logico

La seconda fase della progettazione logica corrisponde a una traduzione tra modelli di dati diversi: *da un diagramma E-R ristrutturato si costruisce uno schema logico.*

La fase di traduzione richiede di eseguire bovinamente alcuni passi definiti, i quali dipendono dal costrutto che stiamo cercando di tradurre:

- **Associazione molti-a-molti**: un'associazione **molti-a-molti** tra due entità può essere facilmente decomposta in tre entità: le due iniziali e una terza relazione che ha come identificatori primari gli identificatori primari delle entità che associava, con due vincoli di integrità referenziale tra gli identificatori.
- **Associazione uno-a-molti**: quando ci si trova a lavorare con due entità legate da un'associazione **uno-a-molti**, è sufficiente inserire l'identificatore dell'entità "unica" nell'altra, se presente, specificando il vincolo di integrità referenziale. Questo stesso procedimento vale anche per entità con *identificatore esterno*; è infatti sufficiente inserire l'identificatore all'interno della relazione che ne fa utilizzo.

- **Associazioni uno-a-uno**: le associazioni **uno-a-uno** hanno molte possibilità di traduzione; una di queste è la creazione di un'entità intermedia che abbia entrambi gli identificatori delle entità dell'associazione, uno dei due identificatori come identificatore primario, l'altro con un vincolo di unicità, specificando i vincoli di integrità necessari.

La convenzione richiede che uno schema logico specifichi le entità nel seguente modo:

NomeEntità(IdentificatorePrincipale, Attributo1, Attributo2, . . . , IdentificatoreEsterno*)**

Documentazione di uno schema logico Come nel caso della modellazione concettuale, il risultato della progettazione logica non è costituito solo da un semplice schema di una base di dati, ma anche da una documentazione ad esso associata. A tale scopo, è stata introdotta una convenzione grafica per specificare, oltre alle entità, anche tutti i vincoli di integrità:

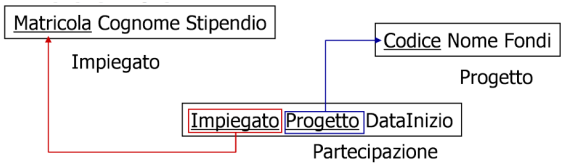


Figure 5.1: Formalismo grafico documentazione E-R

Chapter 6

Normalizzazione e forme normali

0.1 Dipendenze funzionali

Per studiare in maniera sistematica i concetti introdotti all'inizio è necessario far uso di uno specifico strumento: le ***dipendenze funzionali***. Esse rappresentano un particolare **vincolo di integrità** per il *modello relazionale*, utilizzati per specificare legami di tipo funzionale tra gli attributi di ogni relazione.

Il concetto di **dipendenza funzionale** può essere formalizzato come segue

> Data una relazione r su uno schema di relazione R definito su un insieme di attributi X e dati due sottoinsiemi $Y, Z \subset X$, diremo che *esiste su r una dipendenza funzionale tra Y e Z se per ogni coppia di tuple t_1, t_2 in r aventi gli stessi valori su Y, risulta che t_1 e t_2 hanno gli stessi attributi anche su Z.*

Una dipendenza funzionale tra gli attributi di Y e gli attributi di Z viene generalmente indicata come

$$Y \longrightarrow Z \tag{1}$$

Risulta fondamentale notare che, se l'insieme Z è composto dagli attributi $A_1, A_2, \ldots, A_n$, allora una relazione soddisfa $Y \longrightarrow Z$ se e solo se $Y \longrightarrow A_1, A_2, \ldots, A_n$. Inoltre, come ogni altro vincolo, una dipendenza funzionale viene associata ad uno schema: *una relazione su quello schema verrà considerato soddisfatta se e solo se la dipendenza funzionale è soddisfatta.*

Ad esempio, si supponga di avere una relazione strutturata nel seguente modo:

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

Figure 6.1: Schema di relazione d'esempio

Applichiamo la definizione e prendiamo le due tuple che contengono come **Impiegato** Bianchi, osservandole ci si rende conto che ambo le tuple hanno lo stesso valore nell'attributo **stipendio**, stessa cosa vale anche per le tuple in cui l'attributo **Impiegato** è Neri. Questo fatto ci porta ad una conclusione: *esiste una dipendenza funzionale tra l'attributo **Impiegato** e l'attributo **Stipendio***

$$\text{Impiegato} \longrightarrow \text{Stipendio} \tag{2}$$

Inoltre, intuitivamente vale anche che

$$\text{Impiegato Progetto} \longrightarrow \text{Progetto} \tag{3}$$

In effetti, una dipendenza funzionale tra due insiemi di attributi Y, Z dove

$$Z \subseteq Y \tag{4}$$

viene definita come **dipendenza funzionale banale**. Tutte le dipendenze funzionali per cui non vale la condizione sopracitata è una **dipendenza funzionale non banale**. Infine, se prendiamo una chiave K di una relazione r, è facile notare che esiste una **dipendenza funzionale** tra K e ogni altro attributo della relazione. Ad esempio, nello schema di relazione usato precedentemente vale che

$$\text{Impiegato Progetto} \longrightarrow \text{Stipendio Progetto Bilancio} \tag{5}$$

Possiamo quindi concludere che il **vincolo di dipendenza funzionale*generalizza il vincolo di chiave***. Più precisamente, una dipendenza funzionale $Y \longrightarrow Z$ su uno schema $R(X)$ **degenera in vincolo di chiave** se e solo se

$$Y \cup Z = X \tag{6}$$

In altri termini, se Y è una super-chiave, allora essa avrà un vincolo di integrità rispetto ad ogni altro attributo della relazione.

Esistono due tipologie di dipendenze funzionali

- **Dipendenze funzionali parziali**: Si ha una dipendenza funzionale parziale quando $Y \longrightarrow Z$ e inoltre, per ogni possibile $W \subset Y$ esiste almeno un sottoinsieme W_i tale che $W_i \longrightarrow Z$.
- **Dipendenze funzionali complete**: Si ha una dipendenza funzionale completa quando $Y \longrightarrow Z$ e inoltre, per ogni possibile $W \subset Y$ non vale che $W \longrightarrow Z$. Inoltre, se Y è chiave, allora $Y \longrightarrow X$ è una dipendenza funzionale completa.

Le dipendenze funzionali sono fondamentali per verificare la presenza di anomalie all'interno della base di dati e data la loro importanza, per indicare uno *schema di relazione che verifica un insieme di dipendenze funzionali* F verrà utilizzata la notazione

$$R(X, F) \tag{7}$$

Sia F un insieme di dipendenze funzionali definite su $R(Z)$ e sia $X \longrightarrow Y$una dipendenza funzionale, si dice che F implica $X \longrightarrow Y$. La definizione di implicazione non è però **direttamente utilizzabile nella pratica**, essa infatti prevede una quantificazione universale sulle istanze della base di dati e inoltre, non esiste un algoritmo per calcolare tutte le dipendenze funzionali implicate da un insieme F.

A questo scopo **Amostrong** ha fornito delle **regole di inferenza** che permettono di derivare costruttivamente tutte le dipendenze funzionali che sono implicate da un dato insieme iniziale. Queste regole sono

- **Riflessività**: Se $Y \subseteq X$ allora $X \longrightarrow Y$
- **Additività**: Se $X \longrightarrow Y$ allora $XZ \longrightarrow YZ$ per qualunque Z.
- **Transitività**: Se $X \longrightarrow Y$ e $Y \longrightarrow Z$ allora $X \longrightarrow Z$.

0.2 Derivazione delle dipendenze funzionali

Attraverso le leggi di Amstrong è possibile derivare alcune dipendenze funzionali. Dati

- Un **insieme** di regole di inferenza RI.
- Un insieme di dipendenze funzionali F.
- Una dipendenza funzionale f.

Una derivazione di f da F secondo RI è una sequenza finita $f_1, \ldots, f_m$ dove

$$f_m = f \tag{8}$$

e ogni elemento f_i è ottenuto dalle precedenti dipendenze $f_1, \ldots, f_{i-1}$ della derivazione usando una regola di inferenza. Indichiamo inoltre con $F \vdash X \longrightarrow Y$ il fatto la dipendenza funzionale $X \longrightarrow Y$ sia derivabile da F usando RI. Attraverso questo sistema è possibile calcolare alcune **regole di derivazione comuni**:

- **Unione**: $\{X \longrightarrow Y, X \longrightarrow Z\} \vdash X \longrightarrow YZ$
- **Decomposizione**: $\{X \longrightarrow YZ\} \vdash X \longrightarrow Y$.
- **Indebolimento**: $\{X \longrightarrow Y\} \vdash XZ \longrightarrow Y$
- **Identità**: $\{\} \vdash X \longrightarrow X$

1 Chiusura di un insieme di attributi

Dato uno schema $R(T, F)$ con $X \subseteq T$, la **chiusura di uno schema di attributi rispetto a F**, indicata con simbolo X_F^+ è definita come

$$X_F^+ = \{A \in T \mid F \vdash X \longrightarrow A\} \tag{9}$$

O in altre parole, l'insieme degli attributi che dipendono funzionalmente da X rispetto ad un insieme F di dipendenze. Se non vi sono ambiguità scriveremo semplicemente X^+. Inoltre, il teorema di **chisura delgi attributi ci dice che**

$$F \vdash X \longrightarrow Y \iff Y \subseteq X^+ \tag{10}$$

In altre parole, se da un insieme di dipendenze funzionali riesco a derivare che $X \longrightarrow Y$ allora l'attributo Y appartiene alla **chiusura rispetto ad X**.

2 Correttezza e Completezza

Dato un qualche insieme di regole di inferenza RI e un insieme di dipendenze funzionali F, abbiamo che

- **RI** è **corretto** se
$$F \vdash X \longrightarrow Y \implies F \vDash X \longrightarrow Y \tag{11}$$
in altre parole, applicando RI a un insieme F di dipendenze funzionali, si ottengono solo dipendenze logicamente implicate da F.
- RI è **completo** se
$$F \vDash X \longrightarrow Y \implies F \vdash X \longrightarrow Y \tag{12}$$
Applicando RI a un insieme F di dipendenze funzionali, si ottengono tutte le dipendenze logicamente implicate da F.

Le **regole di inferenza di Armstrong** sono corrette e complete, ciò ci permette di scambiare $\vDash$ con $\vdash$ dovunque, anche nella definizione stessa di **chiusura rispetto ad un attributo**

$$X_F^+ = \{A \in T | F \vDash X \longrightarrow A\} \tag{13}$$

Le **regole di inferenza** di Armstrong sono **minimali**, quindi **ignorando** anche solo una di esse, l'insieme di regole che rimangono *non sono più complete.*

3 Chiusura di un insieme di dipendenze funzionali

Sia F un insieme di **dipendenze funzionali** definite su $R_F(X)$. Si definisce come **chiusura di F**, l'*insieme di tutte le* ***dipendenze funzionali*** *implicate da F*

$$F^+ = \{X \longrightarrow Y | F \implies X \longrightarrow Y\} \tag{14}$$

quindi, dato un insieme di **dipendenze funzionali** F definite su $R(x)$, un'istanza $r \in R$ che soddisfa F, allora, essa soddisfa anche le **dipendenze funzionali** di F^+.

Attraverso le **regole di Armstrong** è possibile definire un algoritmo per ottenere la chiusura di un insieme di dipendenze funzionali F.

4 Chiavi

Dato uno **schema di relazione** $R(T, F)$, un insieme di attributi $K \subseteq T$ si dice **super-chiave** di R se la dipendenza funzionale $K \longrightarrow T$ è implicata da F, ovvero se $K \longrightarrow T \in F^+$. Inoltre, un insieme di attributi $K \subseteq T$ si definisce **chiave** di R se e solo se K è **super-chiave** di R e non esiste alcun sottoinsieme proprio di K che sia **super-chiave** di R.

Trovare tutte le chiavi di una relazione $R(Z)$ richiede un algoritmo di complessità esponenziale nel caso pessimo, il quale richiede di passare per alcuni step:

- Porre tutti gli attributi che stanno in tutte le chiavi a sinistra (riferendoci con il nome N a questo insieme).
- Gli attributi posti a destra saranno quelli non presenti in alcuna chiave.
- Si aggiunge a N un attributo alla volta tra tutti quelli che non stanno solo a destra, poi una coppia di attributi e così via, chiamiamo X_i questo sottoinsieme di attributi e ad ogni passo si controlla se la dipendenza $N \cup X_i$ esiste.

L'algoritmo per il calcolo della chiusura di un insieme di attributi può anche essere utilizzato per verificare se un insieme di attributi è **chiave** o **super-chiave**. In particolare, se $X \subseteq T$ è **super-chiave** di $R(T, F)$ allora

- $X \longrightarrow T \in F^+$, quindi, $T \subseteq X^+$

Se $X \subseteq T$ è **chiave**, allora

- $T \subseteq X^+$ e inoltre non esiste alcuna $Y \subset X$ tale che $T \subseteq Y^+$.

5 Copertura di insiemi di dipendenze funzionali

Talvolta potrebbe essere utile sostituire un insieme di dipendenze funzionali F con un altro insieme di dipendenze funzionali G, magari perché più semplice da gestire. Due **insiemi di dipendenze funzionali** F e G, definite sugli stessi attributi X di una relazione $R(X)$ sono **equivalenti** $(F \equiv G)$, quindi **F è una copertura di G**, se e solo se

$$F^+ = G^+ \tag{15}$$

La relazione di equivalenza permette di stabilire se due schemi di relazione rappresentano gli stessi fatti, basta infatti che abbiano gli stessi attributi e le stesse dipendenze funzionali.

È possibile definire dei criteri di **semplicità** per un insieme di dipendenze funzionali

- Un insieme di **dipendenze funzionali** si definisce *non ridondante* se non esiste dipendenza funzionale $f \in F$ tale per cui $F - \{f\} \implies f$.
- Un insieme di **dipendenze funzionali** si definisce *ridotto* se non è *ridondante* e se non esiste un insieme F' equivalente a F ottenuto eliminando attributi dai primi membri di una o più dipendenze di F.

Ad esempio, si considerino tre insiemi di dipendenze funzionali

$$\begin{aligned} F_1 &= \{A \longrightarrow B, AB \longrightarrow C, A \longrightarrow C\} \\ F_2 &= \{A \longrightarrow B, AB \longrightarrow C\} \\ F_3 &= \{A \longrightarrow B, A \longrightarrow C\} \end{aligned} \tag{16}$$

Applicando le definizioni scritte sopra possiamo facilmente osservare che F_1 è **ridondante**, in quanto soltanto con le prime due dipendenze funzionali è possibile ricavare la terza. F_2 è invece **non ridotto**, in quanto, vista la prima dipendenza funzionale, è possibile eliminare B al primo membro della seconda. F_3 è **ridotto**. Da queste due proprietà è possibile introdurre il concetto di **copertura minimale**. Sia F un insieme di ***dipendenze funzionali***, F è una **copertura minimale** se e solo se è **ridotto**.

6 Normalizzazione e decomposizione

Il processo di normalizzazione sfrutta tutto ciò che abbiamo appena descritto per eliminare problematiche riguardanti gli schemi di relazione nella base di dati. Prendiamo come esempio lo schema di relazione usato in precedenza

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

Figure 6.2: Schema di relazione d'esempio

Per questo esempio abbiamo già definito le dipendenze funzionali, sappiamo infatti che

$$\begin{aligned} &\text{Impiegato} \longrightarrow \text{Stipendio} \\ &\text{Progetto} \longrightarrow \text{Bilancio} \\ &\text{Impiegato Progetto} \longrightarrow \text{Funzione} \end{aligned} \tag{17}$$

Osservando lo schema di relazione è possibile giungere a una importante osservazione, *nello schema di relazione vi è una forte presenza di ridondanze*, o per meglio dire, di attributi che si ripetono senza che ve ne sia necessità. Attraverso il processo di decomposizione posso, sfruttando le dipendenze funzionali determinate in precedenza, scomporre lo schema di relazione in tre ulteriori schemi.

Dato uno schema di relazione $R(T)$ e l'insieme di schemi di relazione $R_1(X_1), R_2(X_2), \ldots, R_n(X_n)$ è una decomposizione se e solo se

$$X = \bigcup_{i=1}^{n} X_i \tag{18}$$

La decomposizione è un processo che, talvolta, può essere rischioso, portando a perdita di dati o altre problematiche. Per comprendere meglio questa problematica si può ricorrere ad un esempio

Inoltre, se $\rho = \{R_1(X_1), \ldots, R_n(X_n)\}$ è una **decomposizione** di $R(T, F)$, allora per ogni istanza $r \in R(X)$ si ha

$$r \subseteq \pi_{X_1}(r) \bowtie \ldots \bowtie \pi_{X_n}(r) \tag{19}$$

da questo teorema possiamo giungere ad una conclusione importante: l'informazione viene persa quando **ricostruendo una relazione, otteniamo più tuple che nella relazione originaria**.

Quando si **decompone** uno schema di relazione è fondamentale che esso avvenga*senza perdita*. Preso uno **schema di relazione** $R(X)$ definito su un insieme di attributi X, decomposto in ulteriori schemi di relazione $R_1(X_1), R_2(X_2), \ldots, R_n(X_n)$, definiti su inisiemi di attributi $X_1, X_2, \ldots, X_n$ tale per cui

$$\bigcup_{i=1}^{n} X_i = X \tag{20}$$

abbiamo una **decomposizione senza perdita** se vale la seguente proprietà: dato una $r \in R(X)$ e dati $r_1, r_2, \ldots, r_n \in R_1(X_1), R_2(X_2), \ldots, R_n(X_n)$ vale che

$$r = \bowtie_{i=1}^{n} r_i \tag{21}$$

Esiste anche una condizione che garantisce la **decomposizione senza perdita**. Sia $r \in R(X)$ un'istanza di relazione di uno schema di relazione definito su un insieme di attributi X. Siano inoltre $X_1, X_2, \ldots, X_n$ insiemi di attributi tali per cui

$$X = \bigcup_{i=1}^{n} X_i \tag{22}$$

Sia inoltre X_0 un insieme di attributi definito come

$$X_0 = \bigcap_{i=1}^{n} X_i \tag{23}$$

allora **r si decompone senza perdita** su $X_1, X_2, \ldots, X_n$ se soddisfa una tra le dipendenze funzionali

$$\begin{aligned} X_0 &\longrightarrow X_1 \\ X_0 &\longrightarrow X_2 \\ &\vdots \\ X_0 &\longrightarrow X_n \end{aligned} \tag{24}$$

In altre parole, gli attributi comuni alle n relazioni devono essere chiave di una delle tabelle.

Ad esempio, dato un insieme di attributi **ABC** e le sue *proiezioni* **AB** e **AC**. Supponiamo ora che r soddisfi $A \longrightarrow C$, allora **A** *è chiave per la proiezione di r su* **AC**, quindi, non ci sono valori duplicati di **A** all'interno della *proiezione*. Applicando il join, vengono costruite delle tuple sfruttando le tuple presenti nelle due proiezioni. Consideriamo ora una generica tupla, ottenuta dal join, $t = (a, b, c)$, ottenuta da $t_1 = (a, b)$ nella proiezione di **r** su **AB** e $t_2 = (b, c)$ nella proiezione di **r** su **AC**, nel risultato del **join** e dimostriamo che appartiene ad **r**, provando così l'uguaglianza delle due relazioni. Quindi, per definizione stessa dell'operatore di **proiezione**, devono esistere due tuple in **r**, t'_1 con valori (a, b) su **AB** e t'_2 con valori (a, c) su **AC**. Poiché **r** soddisfa $A \longrightarrow C$ esiste un solo valore su **C** in **r** associato al valore **a** su **A** e dato che (a, b) compare nella proiezione, esso è esattamente **c**. Quindi, il valore di t'_1 su C è proprio così c e così t'_1 ha valori (a, b, c), coincidendo quindi con t.

Oltre alla perdita di dati, un altro problema che si potrebbe verificare durante il processo di decomposizione è la **perdita delle dipendenze funzionali**. Dato $R(T, F)$ e $T_i \subseteq T$, si definisce come **proiezione dell'insieme di dipendenze F sull'insieme di attributi** T_i

$$\pi_{T_i}(F) = \{X \longrightarrow Y \in F^+ | X, Y \subseteq T_i\} \tag{25}$$

quindi l'insieme delle dipendenze funzionali relative agli attributi presenti nella proiezione. Dato uno schema di relazione $R(T, F)$ e una decomposizione $\rho = \{R_1(X_1), \ldots, R_n(X_n)\}$, si dice che ρ è una decomposizione di $R(T, F)$ che **preserva le dipendenze** se e solo se

$$\bigcup_{i=1}^{n} \pi_{T_i}(F) = F \tag{26}$$

7 Forma normale di Boyce-Codd

Dato uno schema di relazione $R(T, F)$, si dice che una qualsiasi **instanza di relazione** $r \in R$ è in **forma nomrmale di Boyce-Codd** se e solo se, per ogni dipendenza funzionale non banale $X \longrightarrow A \in F^+$, X contiene una chiave K di r, cioè se X è una **super-chiave** di R. L'idea su cui si basa la **BCNF** è che una dipendenza funzionale $X \longrightarrow A$, in cui X non contiene attributi estranei, indica che, nella realtà che si modella, esiste una collezione di entità omogenee che sono identificate univocamente da X.

Per determinare se una relazione $R(T, F)$ è in **BCNF** possiamo sfruttare un semplice teorema

> Uno schema di relazione $R(T, F)$ è in BCNF se e solo se per ogni **dipendenza funzionale non banale** $X \longrightarrow A \in F$, X è una **super-chiave**.

da questo teorema è possibile ricavarne un corollario. Uno schema di relazione $R(T, F)$ con **F copertura minimale** è un BCNF se e solo se per ogni **dipendenza funzionale elementare** $X \longrightarrow A \in F$, X è una **super-chiave**. Calcolare F^+ richiede, purtroppo, un algoritmo di complessità esponenziale, invece, la verifica di una forma BCNF un algoritmo di complessità polinomiale.

Prendiamo come esempio la relazione già usata all'interno di questo paragrafo

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

Figure 6.3: Schema di relazione d'esempio

Osserviamo che, nell'esempio, le due proprietà che causano anomalie corrispondono esattamente ad attributi coinvolti in dipendenze funzionali. In particolare esse sono

$$\begin{aligned} \textbf{Impiegato} &\longrightarrow \textbf{Stipendio} \\ \textbf{Progetto} &\longrightarrow \textbf{Bilancio} \end{aligned} \tag{27}$$

Inoltre, l'attributo **funzione** può essere modellato per mezzo di una dipendenza funzionale, infatto esso è univoco per ogni coppia di attributi **Progetto**, **Impiegato**

$$\textbf{Progetto Impiegato} \longrightarrow \textbf{Funzione} \tag{28}$$

Notiamo che la terza dipendenza funzionale generi valori univoci per ogni possibile tupla, mentre invece le prime due dipendenze funzionali, generino invece delle anomalie e delle ridondanze. Possiamo quindi concludere che le ridondanze e anomalie sono causate da dipendenze funzionali $X \longrightarrow A$ tali per cui **X non è una chiave**.

8 Terza forma normale

Nella maggior parte dei casi si può raggiungere una decomposizione ottimale anche soltanto la forma normale di **Boyce e Codd**. Talvolta però questo non è possibile, si consideri l'esempio seguente

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

Figure 6.4: Esempio

da questa relazione è possibile ricavare le seguenti dipendenze funzionali

$$\begin{aligned} \text{Dirigente} &\longrightarrow \text{Sede} \\ \text{Progetto Sede} &\longrightarrow \text{Dirigente} \end{aligned} \tag{29}$$

Si nota facilmente che la relazione non è in forma normale di **Boyce e Codd**, in quanto, presa ogni possibile superchiave, esiste una dipendenza funzionale $f \in F$ tale per cui la candidata superchiave non è **superchiave**.

Per trattare casi come questo si ricorre a una forma normale **meno restrittiva** di quella di Boyce e Codd, che sostanzialmente consente situazioni come quella descritta, ma non ammette ulteriori fonti di ridondanza e anomalia. Una relazione R(T,F) è in **terza forma normale** se e solo se, per ogni dipendenza funzionale $X \longrightarrow A \in F^+$ definita su di essa, vale almeno una delle seguenti condizioni

- X contiene una chiave K di r.
- A appartiene ad almeno una chiave di r (A è un **attributo primo**)

ritornando al nostro esempio, possiamo notare che la dipendenza funzionale

$$\text{Progetto Sede} \longrightarrow \text{Dirigente} \tag{30}$$

ha come primo membro una chiave di r. Inoltre, la dipendenza funzionale

$$\text{Dirigente} \longrightarrow \text{Sede} \tag{31}$$

pur non contenendo chiavi al primo membro, ha come secondo membro un unico attributo a secondo membro che fa parte della chiave **Progetto Sede**.

Come abbiamo avuto modo di osservare, la **terza forma normale è tendenzialmente più debole della forma normale di Boyce e Codd**, ma d'altra parte, è sempre ottenibile. Infatti è possibile dimostrare che ogni relazione è decomponibile senza perdita in terza forma normale.

8.1 Sintesi di uno schema in 3FN

Il problema di decidere se uno schema di relazione è in terza forma normale appartiene alla classe dei problemi **NP-Completi**, infatti, il miglior algoritmo deterministico noto ha complessità esponenziale nel caso peggiore. L'algoritmo si basa su un intuzione

> Dato un insieme T e una copertura minimale G, si divide G in gruppi G_i in modo che tutte le dipendenze funzionali di ogni gruppo G_i abbiano la stessa **parte sinistra**.

Successivamente da ogni gruppo G_i si definisce uno schema di relazione composto da tutti gli attributi che appaiono in G_i, la cui chiave, detta **chiave sintetizzata** è la **parte sinistra comune**. L'algoritmo può essere sintetizzato in modo schematico

- Viene creata una copertura minimale G di F.
- Viene partizionato G in tanti piccoli sottoinsiemi di dipendenze funzionali tali per cui il membro sinistro è lo stesso.
- Vengono creati degli schemi di relazione, per ogni G_i, in cui compaiono tutti gli attributi coinvolte nelle dipendenze funzionali $g \in G_i$.
- Se uno dei sottoschemi di relazione è propriamente contenuto in un altro, allora procedo a rimuoverlo.
- Viene costituito uno schema di basi di dati con uno schema di relazione $R_i(U_i)$ per ciasciun elemento $U_i \in U$ con associate le dipendenze in G i cui attributi sono tutti in U_i.
- Se nessuno degli U_i costituisce una chiave per la relazione originaria $R(U)$, allora viene calcolata una chiave K di R(U) e viene aggiunto allo schema generato al passo precedente uno schema di relazione sugli attributi K.

Per teorema, qualunque sia la relazione, l'esecuzione dell'algoritmo per decomposizione in 3FN su tale relazione termina e produce una decomposizione della relazione tale che:

- La decomposizione prodotta è in 3FN.
- La decomposizione prodotta preserva i dati e le dipendenze funzionali.

9 Progettazione di Basi di Dati e Normalizzazione

La teoria della normalizzazione può essere anche applicata alla progettazione di una Base di Dati, infatti durante la fase di progettazione è normale incorrere in errori e problematiche. Applicando però la teoria della normalizzazione è possibile verificare e eliminare queste problematiche prima che esse possano affliggere la nostra base di dati.

9.1 Verifiche di normalizzazione su entità

Le idee alla base della normalizzazione possono essere utilizzate anche durante la fase di progettazione concettuale, con riferimento ai costrutti del modello Entità-Relazione. Lavorare su entità e relazioni risulta estremamente semplice, possiamo infatti considerare ogni relazione come se fosse un entità e procedere con i metodi visti fino ad ora. Prendiamo ad esempio la seguente entità

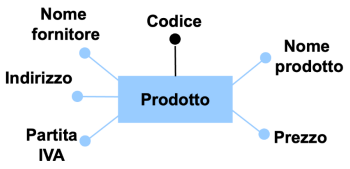


Figure 6.5: Entità di esempio

su cui è definita la seguente dipendenza funzionale

$$\text{Partita IVA} \longrightarrow \text{Nome Fornitore, Indirizzo} \tag{32}$$

Si verifica facilmente che la relazione non è in forma normale di Boyce e Codd, in quanto è possibile trovare una dipendenza funzionale per cui **codice** *non è chiave*. Inoltre, essa viola anche la terza forma normale, in quanto la coppia **Nome Fornitore, Impiegato** non è chiave. Si potrebbe quindi immaginare di avere una decomposizione fatta nel seguente modo

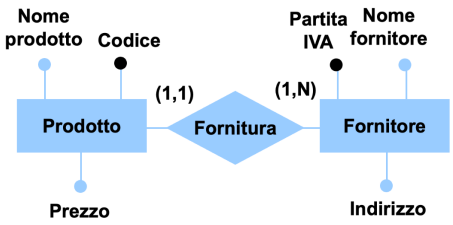


Figure 6.6: Entità decomposta

9.2 Verifiche di normalizzazioni su associazioni

Per quanto riguarda le associazioni il ragionamento è più semplice, in quanto l'insieme di occorrenze di ciascuna associazione è una relazione ed è quindi possibile applicare direttamente i concetti connessi alle forme normali. Per verificare il soddisfacimento della terza forma normale, è necessario individuare le dipendenze funzionali che sussistono tra le entità coinvolte. Ovviamente, è facile notare che ogni associazione binaria è di base in **terza forma normale**, ha senso quindi fare la verifica soltanto sulle associazioni che coinvolgano al più tre entità. Consideriamo, ad esempio, la seguente associazione

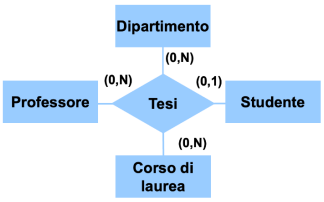


Figure 6.7: Associazione d'esempio

Nella quale ho individuato le seguenti dipendenze funzionali

$$\begin{aligned} &\textbf{Studente} \longrightarrow \textbf{Corso Di Laurea} \\ &\textbf{Studente} \longrightarrow \textbf{Professore} \\ &\textbf{Professore} \longrightarrow \textbf{Dipartimento} \end{aligned} \tag{33}$$

Ci accorgiamo che la chiave unica è **studente** e che, di conseguenza, la tabella viola la terza forma normale. Anche logicamente possiamo arrivare alla conclusione che l'afferenza di un professore al dipartimento non dipende dal fatto che vi siano studenti o meno che discutono una tesi. Possiamo decomporre l'associazione come segue

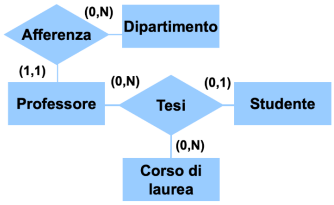


Figure 6.8: Associazione dell'esempio normalizzata

Lo schema è adesso in terza forma normale, nonostante ciò è ancora possibile fare delle considerazioni su di esso. Le proprietà descritte dalle due dipendenze funzionali sono **indipendenti**, non tutti gli studenti stanno svolgendo una tesi. A questo problema si può ovviare utilizzando dei valori nulli, ma a livello di progettazione concettuale occorre distinguere i due concetti. Anche in questo caso è legittimo attuare un processo di decomposizione, andando a definire la relazione come segue

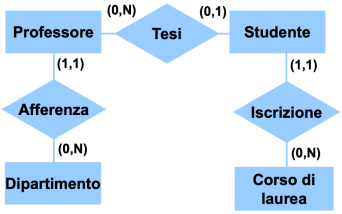


Figure 6.9: Ulteriore decomposizione della relazione

Part III

Funzionamento di una base di dati

Chapter 7

Gestione interna di una Base di Dati

Per spiegare in maniera coerente e comprensibile l'organizzazione interna di una base di dati e la gestione delle interrogazioni è necessario tenere bene a mente una figura

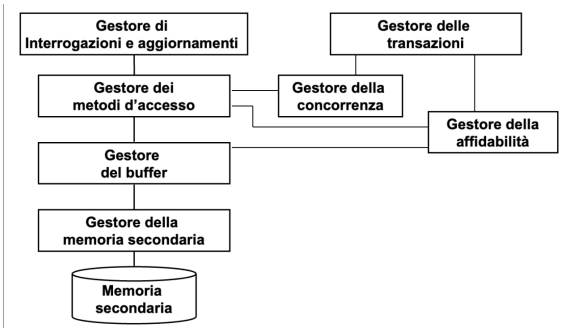


Figure 7.1: Organizzazione interna di una base di dati

Nelle applicazioni per Basi di Dati le operazioni vengono specificate mediante linguaggio SQL, affidate a un modulo detto **gestore delle interrogazioni**, che le traduce in forma interna, per passarle a un modulo detto **ottimizzatore delle interrogazioni** che ha lo scopo di *rendere più efficenti le interrogazioni*, successivamente realizzarle in termini di operazioni di livello più basso, le quali fanno riferimento alla struttura fisica dei file e sono gestite da un modulo detto **gestore dei metodi d'accesso**. Quest'ultimo modulo, conoscendo i dettagli della struttura fisica dei file, trasforma le richieste in operazioni di **lettura** o **scrittura** di dati in memoria secondaria, che vengono peò mediate da un modulo, chiamato **gestore dei buffer**, che ha la responsabilità di mantenere temporaneamente porzioni di basi di dati in memoria centrale, per favorirne l'efficienza, mantenendone comunque **integrità** e **sicurezza**. Il gestore dei buffer affida infine al **gestore della memoria secondaria** le richieste effettive di **scrittura** e lettura **fisica**.

1 Memoria secondaria

Il motivo per cui quest'area di memoria prende il nome di **memoria secondaria** è data dal fatto che essa non *è direttamente utilizzabile dai programmi*: i dati, per poter essere utilizzati, devono essere prima trasferiti in **memoria centrale**.

All'interno della memoria secondaria è spesso possibile **trascurare i costi di tutte le operazioni che non siano accessi in memoria secondaria**, di conseguenza, il costo di un operazione qualsiasi può essere valutato solo in base a quante volte accede a quest'area di memoria. Nonostante questo però, il costo di accesso alla memoria secondaria non è sempre costante, esso dipende da tre fattori

- Il **tempo di posizionamento** della testina: il tempo che la testina impiega per raggiungere la traccia di interesse,
- Il **tempo di latenza**: il tempo per accedere al primo byte del blocco di interesse.
- Il **tempo di trasferimento**: tempo necessario a muovere tutti i dati del blocco.

2 Gestione dei Buffer

All'interno di una **Base di Dati**, l'interazione tra la **memoria centrale** e la **memoria secondaria** è realizzata attraverso una preposta, grande zona di memoria centrale detta **buffer** gestita in modo condiviso dal DBMS per tutte le applicazioni.

Il buffer è organizzato in **pagine**, la cui dimensione equivale a quella di un numero intero di blocchi. In questa sezione assumeremo che una pagina abbia la stessa dimensione di un blocco di memoria secondaria. Il modulo chiamato **gestore dei buffer** si occupa del caricamento o dello scaricamento delle pagine dalla memoria centrale alla memoria di massa. Possiamo immaginarci il gestore di buffer come un modulo che, riceve dai programmi richiesta di lettura e scrittura di blocchi ed eseguite le effettive richieste per la lettura o la scrittura sulla base di dati, secondo una tempistica che non coincide necessariamente con quella delle richieste ricevute. In particolare:

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

In entrambi i casi abbiamo ridotto il tempo di risposta di un'applicazione, evitando operazioni sulla memoria di massa.

Le **politiche di gestione del buffer** si comportano in modo molto simile a quelle usate dai sistemi operativi e si basano sull'idea di **località dei dati**, in base al quale i dati referenziati di recente hanno maggior probabilità di essere referenziati nuovamente nel futuro. Inoltre, esiste una regola empirica secondo la quale

> Il 20% dei dati è tipicamente acceduto dall'80% delle applicazioni.

questa legge ha come conseguenza che le pagine richieste più spesso permanono nel buffer per più tempo.

L'interfaccia offerta dal Buffer ai moduli di livello più alto è più complessa di quanto non sembri, infatti essa non può limitarsi alle richieste di lettura e scrittura, ma ha bisogno di informazioni sul prevedibile riutilizzo delle pagine e sull'eventuale necessità di effettuare immediatamente gli aggiornamenti. Per la gestione dei buffer è quindi necessario mantenere informazioni sul loro uso, che possiamo schematizzare come segue:

- Un direttorio descrive il contenuto corrente del buffer indicando per ciascuna pagina quali sono il file fisico e il numero di blocco a essa corrispondente,
- Per ogni pagina del buffer il gestore mantiene alcune *variabili di stato* tra cui: un **contatore** per indicare quanti programmi utilizzano la pagina e un **bit di stato**, per indicare se la pagina è stato modificata.

Ad esempio, passiamo ad illustrare un possibile insieme di operazioni fondamentali allo scopo

- La primitiva **fix**: questa primitiva viene utilizzata dalle **transazioni** per richiedere l'accesso a una pagina; essa restituisce al modulo chiamante il riferimento alla pagina del buffer, in modo che esso possa accedere effettivamente ai dati.
- La primitiva **setDirty**: questa primitiva indica al gestore del buffer che una pagina è stata modificata, causando una modifica del bit di stato rispettivo.

- La primitiva **unfix**: questa primitiva indica la gestore del buffer che il modulo chiamante ha terminato di utilizzare la pagina; come effetto, viene decrementato il contatore di utilizzo della pagina.
- La primitiva **force**: questa primitiva trascrive in memoria secondaria, in modo sincrono, una pagina del buffer.

La scrittura in memoria secondaria può avvenire in modo sincrono, all'interno dell'applicazione che la richiede, attraverso la primitiva **force** oppure in modo **asincrono** a seguito di decisioni del **gestore dei buffer** finalizzate al recupero della memoria o all'ottimizzazione. Potrebbe capitare che una pagina, utilizzata da molte applicazioni, rimanga per lungo tempo nel buffer, subendo anche diverse modifiche, per poi essere trascritta nel buffer con una singola operazione di scrittura.

3 DBMS e File System

Il **file system** è un modulo messo a disposizione dal sistema operativo, i DBMS nel utilizzano le funzionalità. Tuttavia, la relazione che esiste tra le operazioni eseguite dal file system e quelle eseguite dal DBMS non è così immediata, anzi, nonostante i DBMS facciano uso dei file system (cosa che non avveniva fino a qualche tempo addietro), essi mantengono comunque una propria astrazione dei file per garantire **efficienza e transazionalità**. Nei DBMS moderni, l'appoggio che il sistema richiede dai sistemi operativi è limitato alle sole operazioni di eliminazione di file, lettura e scrittura di file, indipendentemente da che essa sia su **blocchi singoli** o **blocchi congiunti**.

Il DBMS gestisce i blocchi dei file allocati come se fossero un unico grande spazio di memoria secondaria e costruisce, in tale spazio, le strutture fisiche con cui implementa le relazioni. Nel caso più ricorrente, ogni blocco è dedicato a tuple di un'unica relazione, ma esistono tecniche che prevedono la memorizzazione delle tuple di più tabelle, tra loro correlate, negli stessi blocchi.

Chapter 8

Gestione delle transazioni

Un criterio che permette di classificare il sistema di badi di dati è **il numero di utenti che possono usufruire simultaneamente del sistema**:

- Un DBMS è **multiutente** se più utenti possono accedere e utilizzare contemporaneamente la base di dati.
- Un DBMS è **monoutente** se al massimo un utente per volta può utilizzare la base di dati.

Nel primo caso, quello della **base di dati multiutente**, l'accesso alla base di dati,in simultanea, è possibile attraverso al concetto di **multitasking**.

Una **transazione** rappresenta un **unità elementare di lavoro** svolta da un applicazione, a cui si vogliono associare particolari caratteristiche di **correttezza**, **robustezza** e **isolamento**. Una **transazione** può essere anche vista come una **successione di comandi** che forma un'**unità logica** di elaborazione della base di dati. Una **transazione** comprende una o più **operazioni di accesso** alla base di dati:

- **Inserimento**.
- **Modifica**.
- **Cancellazione**.

Le quali possono essere **atomizzate** in delle operazioni atomiche di **lettura** e **scrittura**. Una **transazione**, in termini formali, è una **parte di programma** caratterizzata da un **inizio** e una **fine**, al cui interno deve essere eseguita, **una sola volta**, una tra i due comandi: **commit**, **rollback**. Il primo indica che la **transazione stata eseguita correttamente**, il secondo comandi indica che la **transazione è fallita ed è necessario abortire**. Un sistema si definisce **transazionale** se e solo se è in grado di definire ed eseguire transazioni per conto di un certo numero di **applicazioni concorrenti**. Una transazione può trovarsi in diversi stati Una **transazione**

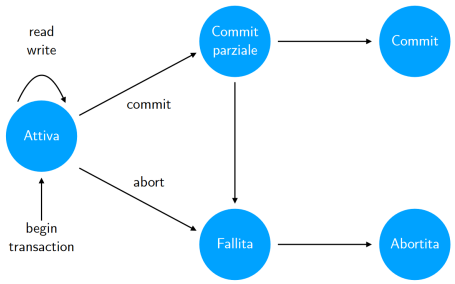


Figure 8.1: Stato di una transazione

entra nello stato **attivo** subito dopo essere iniziata, dove rimane per tutta l'esecuzione. Successivamente si sposta nelllo stato di **commit parziale** quando termina. Infine, esistono tre possibili stati in cui si può trovare

- Una **transazione** si trova nello stato di **commit** se è stata eseguita con successo e tutti i suoi aggiornamenti alla base di dati sono permanenti.
- Una **transazione** si trova nello stato di **fallito** se non può passare dallo stato di **commit parziale** a quello di **commit** o se è stata interrotta in seguito a un falllimento mentre era nello stato attivo.
- Una **transazione** si trova nello stati di **abortito** se è stata interrotta e tutti i suoi aggiornamenti alla base di dati sono stati annullati.

All'interno della base di dati avviene un **controllo dell'affidabilità**, il cui scopo è garantire due proprietà delle **transazioni**:

- ***Atomicità***
- ***Persistenza***

In altre parole, esso garantisce che le **transazioni** vengano eseguite per *intere*, senza che alcune parti vengano tralasciare e che gli effetti di ogni **transazione** siano mantenuti in modo **persistente** sulla base di dati. Il **controllore dell'affidabilità**, per adempiere a questo compito, utilizza uno strumento, chiamato ***log***; Il log è un **archivio** all'interno del quale vengono registrate le azioni eseguite sul **DBMS**. Un **log**, per fare un paragone fiabesco, è come le briciole di pane della favola di **Hansel e Gretel**.

Abbiamo definito prima le proprietà fondamentali delle transazioni, le quali possono essere racchiuse nell'acronimo **ACID**. Ognuna di queste proprietà può essere formalizzata:

- ***A*tomicità**: Una **transazione** è una **unità atomica di elaborazione** e di conseguenza **non può lasciare la base di dati in uno stato intermedio**. Un guasto o un errore, se riscontrato prima del **commit**, dovrebbe causare l'**annullamento** delle operazioni svolta attraverso la direttiva **UNDO**, se invece l'errore viene riscontrato dopo il **commit**, non vi devono essere conseguenze; se necessario vanno ripetute le operazioni. Quest'operazione può essere svola attraverso la direttiva **REDO**.
- ***C*onsistenza**: La **transazione** deve rispettare i **vincoli di integrità**, di conseguenza, alla fine delle **transazioni** la base di dati deve essere in uno stato **coerente**.
- ***I*solamento**: la **transazione non deve risentire degli effetti delle altre transazioni concorrenti**.
- ***D*urabilità**: gli effetti di una **transazione** andata in **commit** non vanno *perduti*.

Transazioni in SQL

Una **transazione** in SQl può essere scritta nel seguente modo

```
START TRANSACTION;
UPDATE ContoCorrente
    SET Saldo = Saldo + 10 WHERE NumConto = 1221;
UPDATE ContoCorrente
    SET Saldo = Saldo - 10 WHERE NumConto = 1342;
SELECT Saldo AS A
    FROM ContoCorrente
    WHERE NumConto = 2342;
IF (A >= 0) THEN COMMIT WORK;
ELSE ROLLBACK WORK;
```

1 Controllo dell'affidabilità

Il controllore dell'affidabilità è responsabile di realizzare i comandi **transazionali** e di realizzare le operazioni di ripristino dopo un malfunzionamento. Queste operazioni di **ripristino** sono divise in due gruppi

- Ripresa a caldo.
- Ripresa a fredddo.

Il controllore dell'affidabilità riceve richieste di accessi a pagine in lettura e scrittura, che inoltra al buffer manager e genera altre richieste di lettura e scrittura di pagine necessarie a garantire la **robustezza** e la **resistenza** ai guasti. Infine il controllore dell'affidabilità predispone tutte i dati necessari alla gestione dei guasti, realizzando operazioni di **checkpoint** e **dump**.

Per poter operare il controllore dell'affidabilità deve avere una **memoria stabile**, quindi una memoria che sia **definitivamente resistente ai gusti**, i quali, se occorrono, sono **eventi catastrofici e imprevedibili**. Il fatto che la **memoria stabile** sia esente da rischi è una pura astrazione, in quanto una minima probabilità esiste, nonostante tramite dei meccanismi di controllo dell'integrità sia possibile portarla quasi a 0.

1.1 Organizzazione del Log

Un **log** è un *file sequenziale*, di cui è responsabile il **controllore dell'affidabilità**, memorizzato all'interno della **memoria stabile** e contenente l'informazione ridondante che permette di ricostruire il *contenuto della base di dati* a seguito di **guasti**. Sul ***log*** vengono registrate le azioni svolte dalle varie transazioni, nell'ordine temporale di esecuzione. Pertanto, il ***log*** ha un **blocco corrente**: l'ultimo blocco a essere stato allocato. I record all'interno del ***log*** sono divisi in due tipologie:

- **record di sistema**:
- **record di transazione**: questi record registrano tutte le attività svolte da ciascuna **transazione**, nell'ordine temporale di esecuzione. Pertanto ogni transazione inserisce nel ***log*** un record di **begin**, vari record relativi alle operazioni della transazione e infine un record di **commit** o di **abort**.

I record di log, che vengono scritti per descrivere le attività di una transazione t_1 sono elencati di seguito.

- I record di **begin**, **commit** e **abort**, che contengono, oltre all'indicazione del tipo di record, anche l'identificativo T della trasazione.
- I record di **update**, che contengono l'indentificativo T della transazione, l'identificativo O dell'oggetto su cui avviene l'update, e poi due valori BS (*Before State*) e AS (After State) che descrivono rispettivamente il valore dell'oggetto O antecedente alla modifica (BS) e il valore successivo alla modifica (AS).
- I record di **insert** e **delete** sono simili a quelli di **update**, da cui si differenziano per l'assenza delle due valori BS e AS rispettivamente.

Tutte le operazioni appena descritte adottano una convenzione sintattica, le operazioni di **begin**, **commit** e **abort** si indicano rispettivamente con $B(T)$, $C(T)$ e $A(T)$, l'operazione di **update** si identifica come $U(T, O, BS, AS)$ e le due operazioni di **insert** e **delete** si identificano come $I(T, O, BS)$ e $D(T, O, AS)$. Questi record consentono di rifare e disfare tutte le operazioni sulla base di dati attraverso le direttive ***undo*** e ***redo***

- ***undo***: **direttiva utilizzata per disfare un'azione su un oggetto O**. Per attuare tale operazione è sufficiente ricopiare nell'oggetto O il valore di **BS**; l'**insert** viene disfatto eliminando l'oggetto O.

- ***redo***: **direttiva utilizzata per ripetere un'azione su un oggetto O**. Per attuare tale operazione è sufficiente ricopiare nell'oggetto O il valore di **AS**; il **delete** viene rifatto eliminando l'oggetto O

Dato che le primitive di ***undo*** e ***redo*** sono definite tramite un'azione di copiatura vale per esse una proprietà, detta **idempotenza**: eseguendo un numero arbitrario di ***undo*** e ***redo*** della stessa azione equivale allo svolgimento di tale azione solo una volta.

Checkpoint e Dump Tutti i record di log presenti nel sistema potrebbero essere sufficenti per ricostruire la base di dati in caso di guasti. Questa modalità di recuperò è però estremamente lunga, in quanto dovrebbe utilizzare l'intero log; per semplficarla sono stati inventati degli opportuni accorgimenti:

- ***checkpoint***: un ***checkpoint*** è un operazione che viene svolta periodicamente dal gestore dell'affidabilità, con l'obiettivo di registrare quali transazioni sono attive. Esistono diverse versioni di questa operazione, nel nostro caso ci limitermo ad analizzarne una:

  1. Si sospende l'accettazione di operazioni di scrittura, di commit o di abort da parte di ogni transazione.
  2. Si trasferiscono in memoria di massa tutte le pagine del buffer su cui sono state eseguite modifiche da parte di transazioni che hanno già effettuato il commit.
  3. Si forza in scrittura nel log un record di checkpoint che contiene gli identificatori di tutte le transazioni attive.
  4. Si riprendono ad accettare tutte le operazioni da parte delle transazioni.

- ***dump***: un ***dump*** è una copia completa e consistente della base di dati, che viene normalmente effettuata in mutua esclusione con tutte le altre transazioni quando il sistema non è operativo (di notte ad esempio). Questa copia è salvata sulla memoria stabile e viene detta ***backup***. Al termine dell'operazione viene registrata sul log un record di dump, il quale segnala la presenza di un ***backup*** a un certo istante.

Le transazione sceglie in modo **atomico** e **indivisibile**, l'esito del **commit** nel momento in cui scrive sul log il record di **commit**. Prima di questa azione, un eventuale guasto porterebbe un **undo** delle operazioni, ricostruendo così lo stato della base di dati. Dopo l'azione di commit, un eventuale guasto porterebbe ad un operazione di redo di tutte le operazioni effettuate, in modo da ricostruire con certezza lo stato finale delle transazioni.

Esecuzione delle transazioni

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

- La regola WAL (*Write Ahead log*): questa regola impone che la parte BS dei record di un log venga scritta nel log prima di effettuare la corrispondente operazione sulla base di dati. Questa regola consente di disfare le scritture già effettuate in memoria di massa da parte di una transazione che non ha ancota effettuato un commit, poiché per ogni aggionrnamento, viene reso disponbile il valore precedente la scrittura.

- La regola del **commit-precedenza**: questa regola impone che la parte AS dei record di un log venga effettuata prima di effettuare il commit. Questa regola consente di rifare le scritture già decise da una transazione che ha effettuato il commit ma le cui pagine modificate non sono ancora state trascritte in memoria di massa.

In realtà, anche se le regola fanno riferimento a *before state* e *after state* dei record di log, nella pratica entrambe le componenti dei record di log vengono scritte assieme, una versione semplificata della WAL ci dice infatti che *i record di log devono essere scritti prima dei corrispettivi della base di dati.*

Le due regole di WAL e di commit precedenza impongono alcune restrizioni ai protocolli per la scrittura dei log e della base di dati, ma lasciano aperte varie possibilità. Supponiamo che l'azione svolta dalle transazioni sia l'**update**, prendiamo ora com esempio alcuni schemi

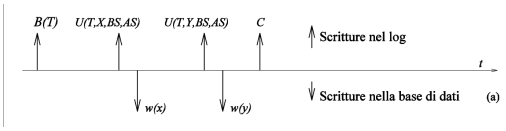


Figure 8.2: Schema 1

Nel primo schema abbiamo che la transazione scrive inizialmente il record B(T), il quale indica l'inizio delle operazioni, successivamente esegue un'azione di **update** scrivendo prima il record di **log** e successivamente la pagina della base di dati, che così passa dal valore BS al valore AS.
Nel secondo schema

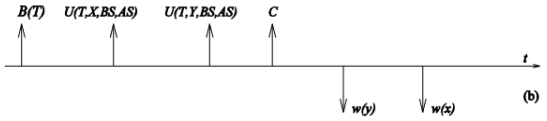


Figure 8.3: Schema 2

la scrittura dei record di log precede quella delle azioni sulla base di dati, che però avvengono dopo la decisione di commit e la conseguente scrittura sincrona del record di commit sul log. Questo schema non richiede operazioni di **undo**. Infine abbiamo l'ultimo schema

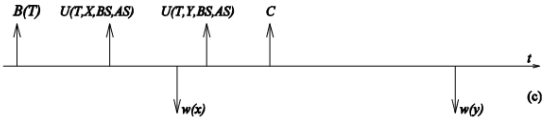


Figure 8.4: Schema 3

Il terzo schema, che è anche quello più generale e comunemente utilizzato, ha come principio che le scritture su una base di dati, una volta protette dalla scrittura del record sul log, possono avvenire in un qualunque momento rispetto alla scrittura del record commit sul log.

La terza modalità viene preferita rispetto alla seconda in quanto, nonostante sia più efficiente nel recovery, è complessivamente meno efficente di una situazione in cui si può gestire liberamente quando operare sulla base di dati

Gestione del guasti

Dal punto di vista del DBMS i guasti si dividono in due tipologie

- **Guasti di *sistema***: sono guasti indotti da *errori di software*. Si traducono in una perdita del contenuto della memoria di massa.
- **Guasti di *dispositivo***: sono guasti relativi ai dispositivi di gestione della memoria di massa. Data la nostra assunzione riguardo alla stabilità della memoria dei log, errori di questo tipo si traducono nella perdita di contenuto della base di dati, ma non dei log.

Il modello ideale di gestione dei guasti è detto *fail-stop*, quando il sistema individua un guasto, sia di sistema che di dispositivo, esso forza un immediato arresto completo delle trasmissioni e il successivo ripristino del corretto funzionamento del sistema operativo (boot). Quindi, viene attivata una procedura di ripresa, chiamata **ripresa a caldo** nel caso di guasto di sistema o, chiamata **ripresa a freddo** nel caso di un guasto di dispositivo. Al termine delle procedure di

ripresa il sistema torna ad essere disponibile, il buffer è vuoto e le transazioni ripartono la loro esecuzione; queste transazioni vanno valutate in base allo stato che avevano prima del guasto:

- se la transazione è **completata** e quindi salvata in memoria stabile, non ci preoccupiamo.
- se la transazione è in **commit**, ma non necessariamente completata, potrebbe essere necessario lanciare n **redo**.
- se la transazione non è in commit va annullata **undo**.

Il restart del sistema si articola in tre passi: viene letto sul file di restart l'indirizzo dell'ultimo checkpoint, vengono preparati due file: un **undo list** con gli identificatori delle transazioni attive e una **redo list** inizialmente vuota. Grazie a questo metodo il guasto rappresenta solo un evento istantaneo che accade a un certo istante dell'evoluzione dela base di dati.

Passiamo ora a capire cosa siano la **ripresa a caldo** e la **ripresa a freddo**.

- Ripresa a caldo: La ripresa a caldosi articola in 4 fasi:
  - Si accede all'ultimo blocco del log e si ricorre all'indietro fino a trovare un checkpoint.
  - Si popolano gli insiemi di **undo** e **redo**, iniziando a eliminare quelle in **undo**.
  - Si ripercorre indietro il lgo disfacendo le transazioni nel set di **undo**, risalendo fino alla transazione più vecchia nei due insiemi.
  - Si ripercorre il log in avanti rifacendo tutte le operazioni nel set di **redo**.
- Ripresa a freddo: La ripresa a freddo si articola in 3 fasi
  - Durante la prima fase si accede al **dump** e si ricopia selettivamente la parte deteriorata della base di dati. Si accede anche al più recente record di dump nel log.
  - Si ripercorre in avanti il log, applicando relativamente alla parte deteriorata della base di dati le azioni presenti sul log.
  - Si opera una ripresa a caldo

2 Controllo di concorrenza

un DBMS deve spesso serviare diverse applicazioni e rispondere alle richieste di più utenti e programmi contemporaneamente. Un unità di misura che viene solitamente utilizzata per misurare il carico applicativo su un DBMS è il **numero di transazioni al secondo** (abbreviato: *tps*) che devono essere elaborate dal DBMS per gestire le applicazioni. In determinati casi, il valore dei ***tips*** può arrivare a migliaia, di conseguenza all'interno di una **base di dati** devono essere implementati dei meccanismi di **Gestione della concorrenza**.

Analogamente allo ***scheduler*** presente su un sistema operativo, il **gestore della concorrenza** ha come compito quello di ricevere le richieste e gestirle, talvolta ordinandole rispetto a specifiche politiche. L'esecuzione concorrente di varie transazioni rappresenta un problema, in quanto non garantisce la mancanza di anomalie se non gestito correttamente. Esistono 5 tipolgie di anomalie

Perdita di aggioramento Supponiamo di avere due **transazioni** identiche che operano sullo stesso oggetto

$$\begin{aligned} t_1 &: r_1(x), x = x + 1, w_1(x) \\ t_2 &: r_2(x), x = x + 1, w_2(x) \end{aligned} \tag{1}$$

dove $r_1(x)$ denota la lettura del generico oggetto x e $w_1(x)$ la sua scrittura. Se le due transazioni vengono eseguite in sequenza, supponendo $x = 2$ all'inizio, risulterà che alla fine avremmo $x = 4$.

Proviamo a immaginare una possibile gestione della concorrenza diversa da quella sequenziale

$$\begin{array}{rl} & r_1(x) \\ x = x + 1 & \\ & \qquad r_2(x) \\ & \qquad x = x + 1 \\ & \qquad w_2(x) \\ & \qquad \text{commit} \\ & w_1(x) \\ & \text{commit} \end{array} \tag{2}$$

In questo caso, il valore finale di x è 3, abbiamo quindi perso un aggiornamento.

Lettura sporca Supponiamo di avere la stessa transazione dell'esercizio precedente, soltanto che nel caso di t_1 l'operazione viene abortita

$$\begin{array}{l} t_1 : r_1(x), x = x + 1, w_1(x) \\ t_2 : r_2(x), x = x + 1, w_2(x) \end{array} \tag{3}$$

Il valore finale di x al termine dell'esecuzione è due, ma la seconda transazione ha letto il valore 3, che essendo la transazione in **abort**, è come se non fosse mai esisto. L'aspetto critico di questa esecuzione è che t_2 ha letto uno **stato intermedio inconsistente** generato da t_1.

Letture incosistenti Suppponiamo ora che a transazione t_1 svolga solamente operazioni di lettura, ma che ripeta la lettura del dato x in diversi istanti successivi, come descritto dalla seguente esecuzione

$$\begin{array}{rl} & \text{t}_1 : \\ & r_1(x) \\ & r_1(y) \\ & r_1(z) \\ & \qquad \text{t}_2 : \\ & \qquad x = x + 100 \\ & \qquad y = y + 100 \\ & \qquad z = z + 100 \\ & \qquad \text{commit} \\ & \text{t}_1 : \\ & r_1(x) \\ & r_1(y) \\ & r_1(z) \\ s = x + y + z & \\ & \text{commit} \end{array} \tag{4}$$

La transazione t_2 non alterla la somma dei valori e quindi non viola il vincolo di integrità; però la variabile s della transazione t_1, che dovrebbe contenere la somma di x,y,z contiene al termine dell'esecuzione il valore 1100. In altri termini, la transazione t_1 osserva solo in parte gli effetti della transazione t_2, e osserva uno stato che non soddisfa i vincoli di integrità.

Inserimento fantasma Considerimao il caso in cui un valore aggregato venga valutato due volte e tra la prima e la seconda valutazione viene inserito un nuovo studente del primo anno, in tal caso i valori medi potranno differire. Per risolvere questa anomalia è sufficente fare riferimento ai soli vecchi valori presenti nella relazione, trattando il nuovo studente cone un **fantasma**.

2.1 Gestione della concorrenza e scheduler

In SQL è possibile specificare per ciascuna transazione un rispettivo **livello di isolamento**, scegliendo tra quattro possibilità

- *read uncommitted*: permette tutte le anomalie.
- *read committed*: evita la lettura sporca
- *repetable read*: evita tutte le anomalie, tranne l'inserimento fantasma.
- *serializable*: evita tutte le anomalie possibili

I livelli di isolamento diversi da **serializable** vanno usati per **transazioni di sola lettura**; ma in generale, per qualsiasi operazione è sempre consigliato utilizzare il livello di isolamento massimo.

Uno scheduler è un S rappresenta una sequenza di operazioni id ingresso e uscita presentate da transazioni concorrenti. Uno ***schedule S*** è quindi una sequenza del tipo

$$S = r_1(x)r_2(y)w_1(x)w_2(y) \tag{5}$$

Il controllo della concorrenza ha la funzione di accettare alcune scheudle di riutarnne altri, in modo da evitare che si verifichno anomalie. Per questa ragione, il modulo che gestisce il controllo di concorrenza prende il nome di **schedurer**: esso ha il compito di intercettare le operazioni compiute dalle transazioni sulle basi di dati, decidendo per ognuna se accettarla, rifiutarla o posticiparla.

Definiamo **seriale** uno schedule in cui le azioni di ciascuna transazione compaiono in sequenza, senza essere inframezzate da istruzioni di altre transazioni

$$S = r_0(x)r_0(y)w_0(x)r_1(x)r_1(y)w_1(x)r_2(x)r_2(y)w_2(x) \tag{6}$$

L'esecuzione di uno schedule S_1 è **corretto**, e quindi **serializzabile**, quando produce lo stesso risultato prodotto da un qualsiasi schedule seriale S_j delle stesse transazioni. Però, affinché la frase *due schedule producono lo stesso risultato* abbia senso è necessario definire un criterio di equivalenza tra **schedule**. Esistono due definizioni principali di equivalenza:

- **View equivalenza**: Per definire quando **due schedule sono view equivalenti** è necessario definire una relazione che lega coppie di operazioni di lettura e scrittura in uno schedule: diciamo che esiste una relazione **legge-da** tra le operazioni $r_i(x)$ e $w_j(x)$ presenti in uno schedule S se $w_j(x)$ precede $r_i(x)$ in S e non c'è nessun $w_k(x)$ tra $r_i(x)$ e $w_j(x)$. Inoltre, la scrittura $w_j(x)$ in S è detta scrittura finale su x se è l'ultima scrittura su x in S. Dopo aver definito queste due nozioni, possiamo passare alla definizione di **view-equivalenza**

  > Due **schedule** S_i e S_j sono detti **view-equivalenti** ($S_i \equiv S_j$) se hanno la stessa **relazione legge-da** e le stesse **scritture finali su ogni oggetto**.

  Uno schedule è **view-serializable** se è **view-equivalente** ad un qualsiasi schedule seriale. L'insieme di tutti gli **schedule seriali** è detto **VSR**. Verificare la view equivalenza di due **schedule** è un operazione con complessità lineare. Decidere sulla view-serializzabilità di uno schedule è però un problema NP-Completo, ciò ci porta alla conclusione che questa equivalenza non è utilizzabile nella pratica.

- **Conflict-equivalenza**: Uno nozione di equivalenza più facilmente utilizzabile si basa sulla definizione di **conflitto**. Date due azioni a_i e a_j tale che $i \neq j$, si dice che a_i è in **conflitto** con a_j se esse operano sullo stesso oggetto e almeno una di esse è una scrittura. Come nel caso della **view-equivalenza** è possibile definire un concetto di *conflict-equivalenza*

  > Uno **schedule** S_i è **conflict-equivalente** allo schedule S_j se i due schedule presentano le stesse operazioni e ogni coppia di operazioni in conflitto è nello stesso ordine nei due schedule.

Uno schedule sarà quindi **confluict-serializable** se è **conflict-equivalente** a ogni altro schedule. Si può dimostrare che ogni schedule CRS è anche VSR, ma non viceversa. Possiamo determinare se uno schedule è **conflict-serializable** tramite il **grafo dei conflitti**. Il grafo è costruito facendo corrispondere un nodo a ogni transazione e tracciando gli archi orientato da t_i a t_j se esiste almeno un conflitto tra un azione a_i e a_j e si ha che a_i precede a_j. Si può dimostrare che uno schedule CSR se e solo se è il **grafo dei conflitti aciclico**. Anche la **conflict-serializzabilità**, pur nonostante sia più rapidamente verificabile è **inutilizzabile nella pratica**. La tecnica sarebbe efficente se potessimo conoscere il grafico fin dall'inizio, ma così non è, costringendo lo scheduler deve operare **incrementalmente**, cioè ad ogni richiesta di operazione decidere se eseguirla subito oppure fare altro.

Nella pratica si utilizzano delle tecniche che garantiscano la **conflict-serializzabilità** senza dover costruire il grafo. Queste teniche prendono il nome di **lock**. Il principio base di questa tenica è che

> Tutte le operazioni di lettura e scrittura devono essere protette tramite l'esecuzione di opportune primitive: *r_lock, w_lock e unlock*.

Lo **scheduler**, detto anche **lock manager**, riceve una sequenza di esecuzione di queste primitive da parte delle transazioni, e ne determina la correttezza con una semplice *ispezione* di una struttura dati. Nell'esecuzioni di operazioni di lettura e scrittura si devono rispettare i seguenti vincoli

- Ogni operazione di lettura deve essere preceduta da un r_lock e seguita da un *unlock*. Il lock, in questo caso si dice, è **condiviso**, perché su un dato posso essere contemporaneamente attivi più lock di questo tipo.
- Ogni operazione di scrittura deve essere preceduta da un w_lock e seguita da un *unlock*. Il lock, in questo caso, si dice **esclusivo**, in quanto non può coesistere con altri lock sullo stesso dato.

Quando una transazione segue queste regole si definisce come **ben formata rispetto al locking**; quando una transazione deve compiere sia un operazione di scrittura che una di lettura ouò richiedere un solo lock di tipo esclusivo oppure passare da un lock condiviso a uno esclusivo. Quando una richiesta di lock viene concessa, si dice che la risorsa viene ***acquisita*** dalla transazione richiedente, quando invece viene invocata la direttiva ***unlock***, si dice che la risorsa è stata ***rilasciata*** e infine, quando una richiesta non viene accettata, la risorsa viene messa in stato di ***attesa***; stato nel quale rimane fintanto che il lock non viene concesso.

Ogni richiesta di lock è composta esclusivamente dall'identificativo della transazione che fa la richiesta e dalla risorsa che la transazione richiede. La gestione dei conflitti da parte del **lock manager** è attuata sfruttando una tabella, chiamata ***tabella dei conflitti***. All'inteno di questa tabella troviamo All'interno della tabella abbiamo i tre possibili stati di una risorsa, la quale può

Richiesta	Stato della risorsa		
	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 - dipende*	OK → free

Figure 8.5: Tabella dei conflitti

essere libera, bloccata da una lettura o da una scrittura. I tre no rappresentano i conflitti che si possono verificare

- Se ho una richiesta di lettura, se la risorsa è libera o vi è un'altra lettura in corso, non ho conflitti di alcun tipo. D'altra parte, se la risorsa è bloccata da una scrittura il lock concesso alla transazione è esclusivo e di conseguenza di verifica un conflitto del tipo R-W.
- Se uno una richiesta di scrittura, se la risorsa è libera non ho conflitti. D'altra parte, se la risorsa è bloccata da un lock di qualsiasi tipo ho un conflitto del tipo W-W o W-R.

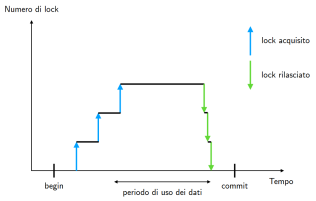


Figure 8.6: Rappresentazione grafica del Locking a due fasi

- Se la richiesta è di **unlock**, abbiamo un errore se la risorsa è libera e abbiamo la risorsa libera nel caso in cui il lock su cui viene invocata la direttiva è di scrittura. Se invece il lock è di lettura, non è detto che dopo aver invocato l'**unlock** la risorsa si liberi, ciò deriva dal fatto che il lock di lettura è condiviso e di conseguenza più transazioni lo possono avere nello stesso momento.

Un algoritmo di **scheduling** utilizzato ampiamente in termini commerciali è il **locking a due fasi**. Questo algoritmo si basa su due semplici regole

- Se una transazione vuole leggere un dato, prima deve acquisire un lock condiviso sul dato.
- Se una transazione vuole scrivere un dato, prima deve acquisire un lock esclusivo sul dato.

Una transazione, dopo aver rilasciato un lock, non ne può acquisire altri. In altre parole

> Una transazione acquisisce tutti i lock di cui ha bisogno e successivamente, man mano che li rilascia non ne può acuisire altri.

Possiamo rappresentare il locking a due fasi nel seguente modo Possiamo dimostrare che ogni schedule che rispetti le condizioni del **locking a due fasi** è anche serializzabile rispetto alla **conflict-equivalenza**. Ammettiamo per assurdo che esista uno schedule $S \in 2PL$ tale che $S \notin CSR$. Se vale l'ultima condizione, vuol dire che sul grafo dei conflitti è possibile trovare un ciclo. Inoltre, se due transazioni sono in conflitto vuol dire che esiste una risorsa x su cui entrambe operano in modo conflittuale e quindi, affinchè A t_2 venga concesso il lock è necessario che t_1 lanci la direttiva di unlock sulla risorsa x. D'altra parte, se esiste un conflitto tra t_1 e t_n vuol dire che entrambe operano in modo conflittuale sulla risorsa y, quindi, affinché t_1 possa operare sulla risorsa y è necessario che t_n invochi la direttiva di unlock sulla risorsa y. Quindi la transazione t_1 non può essere in 2PL, poiché rilascia una risorsa prima di acquisirne un'altra. Si può anche dimostrare che queste due classi non sono equivalenti, infatti possiamo trovare uno scheudule in CSR che non è in 2PL

$$S : w_1(x)r_1(x)r_2(x)w_2(x)r_3(y)w_1(y) \tag{7}$$

notiamo facilmente che la transazione t_1 deve cedere un lock esclusivo sulla risorsa x e successivamente richiederne uno sulla risorsa y.

La 2PL risolve le anomalie di **perdita di aggiornamento**, *aggiornamento fantasma* e di **letture inconsistenti**, ma al contempo può presentare altre anomalie

- ***Cascading Rollback***: il fallimento di una transazione che ha scritto una risorsa deve causare il fallimento di tutte le transazioni che hanno letto il valore scritto.
- ***Deadlock***: due transazioni detengono ciascuna una risorsa e aspettano la risorsa detenuta dall'altra.

Per ovviare a questo problema si introduce un vincolo aggiuntivo, *i lock possono essere rilasciati solo dopo il commit*, quindi solo se la transazione va a buon fine; In questo modo eliminiamo possibilità di avere **letture sporche** e di **cascading rollback**. Un'ulteriore alternativa è data dal controllo basato sui **timestamp**

Controllo di concorrenza basato sui timestamp Questo metodo utilizza il concetto di ***timestamp***, definito come un'*identificatore che definisce un ordinamento totale sugli eventi di un sistema.* Il controllo di concorrenza mediante timestamp avviene nel seguente modo:

- Ad ogni transazione viene associato un ***timestamp***, che rappresenta il momento di inizio e fine della transazione.
- Si accetta uno schedule se e solo se esso riflette l'ordinamento seriale delle transazioni in base al valore del timestamo di ciascuna transazione.

Ad ogni oggetto x vengono assiciati due indici wtm(x) e rtm(x), che sono rispettivamente i timestamp della transazione che ha eseguito l'ultima scrittura e della transazione con t più grande che ha letto x. Allo scheduler arrivano le richieste di accesso agli oggetti del tipo $r_t(x)$ e $w_t(x)$, dove t rappresenta il **timestamp** della transazione che esegue l'operazione. Lo scheduler accetta o meno la richiesta in base a due politiche

- $r_t(x)$: se $t <$ wtm(x) la transazione viene uccisa, altrimenti viene accettata e rtm(x) viene posto al massimo tra t e il suo valore precedente.
- $w_t(x)$: se $t <$ wtm(x) o $t <$ rtm(x) la transazione viene uccisa, altrimenti viene acettata e il valore di wtm(x) viene aggionrato a t.

In pratica, ogni transazione non può leggere o scrivere un dato scritto da una transazione con timestamp superiore e non può scrivere su un dato che è già stato letto da una transazione con timestamp superiore. L'ordine seriale dele transazioni è fissato prima che le operazioni vengano richieste, tutti gli altri ordinamenti non sono accettati. Inoltre, a differenza del 2PL, il TS non può causare **deadlock**, infatti si uccide mediamente una transazione ogni due cnflitti e la probabilità di un deadlok è molto minore rispetto al 2PL.

Risoluzione dei deadlock Supponiamo di avere uno scheduler, il quale si trova in una forma che accetta la presenza di deadlock; per evitare che questo problema insorga è necessario implementare dei meccanismi che possano o prevenire i deadlock, rilveandoli prima che si verifichino. Uno di questi meccanismi è il **grafo delle attese**. All'interno di questo grafo abbiamo due elementi:

- Nodi: le transazioni attive.
- Archi: un arco (T_i, T_j) indica che T_i attende che T_j rilasci la risorsa.

quando all'interno di questo grafo è presente un ciclo, abbiamo una situazione di deadlock. Dopo aver identificato il problema, abbiamo tre possibili strade per risolverlo

- Timeout: Le transazioni rimangono in attesa per un tempo prefissato, se trascorso tale tempo la risorsa non è ancora stata concessa, alla richiesta di lock viene data risposta negativa. In tal modo una transazione in un potenziale stato di deadlock viene tolta dall'attesa e abortita. Questa tenica è quella più utilizzata dalla maggior parte dei sistemi commerciali.
- Rilevamento dello stallo: Ricerca di cicli nel grafo delle attese.
- Prevenzione dello stallo: uccisione di transazioni sospette.

Un problema della tecnica del **timeout** riguarda la scelta dell'intervallo di tempo: un valore troppo elevato tende a risolvere tardi i blocchi critici, lasciando le transazioni per intervalli di tempo molto lunghi in attesa. Un valore troppo basso di timeout rischia di rifiutare transazioni che magari sono solamente in attesa di una risora che si sarebbe liberata di lì a poco. Inoltre, per risolvere dei conflitti e delle possibili situazioni di deadlock abbiamo due politiche

- **Politiche interrompenti**: un conflitto può essere risolto uccidendo la transazione che detiene la risorsa.
- **Politiche non intrrompenti**: una transazione può essere uccisa solo nel momento in cui effettua una nuova richiesta

Una transazione, all'inizio della propria elaborazione, accede ad un oggetto richiesto da molte altre transazoni, così è sempre in conflitto con altre transazioni e, essendo all'inzio del suo lavoro, viene ripetutamente uccisa. Questo stato prende il nome di **starvation**. Per risolvere questa situazione è possibile mantenere invariato il tempo di partenza delle transazioni avortite e fatte ripartire, dando in questo modo priorità alle transazioni piu anziane.

Part IV

Sviluppo di una base di dati

Chapter 9

Introduzione al linguaggio SQL

1 Interrogazioni

Per poter estrarre i dati da una base di dati è necessario compiere delle **interrogazioni**, definite come operazioni sul DBMS che oltre a prelevare i dati ne manipolano il formato. Assumiamo di avere una tabella del tipo

CodFiscale	**Nome**	**Cognome**	**Età**	**AnnoDiNascita**
ABC12345	Mario	Rossi	30	1994
DEF67890	Lucia	Bianchi	25	1999
GHI13579	Giuseppe	Verdi	40	1984

Table 9.1: Esempio di tabella con attributi

compiere un'interrogazione su questa tabella equivale a estrarre un sottoinsieme contenente solo alcune informazioni, prendiamo ad esempio l'interrogazione

> *estrai i **nomi** e i **cognomi** di tutte le persone **maggiorenni***

l'interrogazione, nel modo in cui è stata posta, risulta **ambigua**, il **DBMS** non è infatti in grado di determinare il significato di maggiorenne (poiché cambia da paese a paese) e inoltre, non ho specificato da quale tabella voglio prelevare i dati. Ciò ci porta un'osservazione importante a riguardo delle **interrogazioni** (chiamate anche **query**)

> *è importante definire correttamente i filtri che intendo applicare sulla query, così da evitare ambiguità*

la query che ho definito antecedentemente andrebbe riformulata come

> *estrai i **nomi** e i **cognomi**, dalla tabella **Persone**, di tutte le persone **la cui età è maggiore di diciotto anni***

l'elaborazione di questa query sul DBMS porterebbe ad avere il seguente risultato Il DBMS non è

Nome	**Cognome**
Mario	Rossi
Lucia	Bianchi
Giuseppe	Verdi

Table 9.2: Risultato della query

però in grado di effettuare l'interrogazione in un linguaggio *umano*, ciò ha portato allo sviluppo del linguaggio **SQL** (*Structured Query Language*). La struttura base di una query SQL è la seguente

```
SELECT --attributi da estrarre
FROM --tabella da cui estrarre
WHERE --condizioni per estrarre i dati
GROUP BY --clausola di raggruppamento HAVING
--condizione a riguaro dei raggruppamenti
ORDER BY --clausola di ordinamento
```

Ognuna di queste clausole ha uno scopo preciso

- La clausola **SELECT**, definita in **algebra relazionale** come **proiezione**, permette di specificare quali attributi della relazione dovranno comparire nel risultato.
- La clausola **FROM** permette di specificare da quale tabella prelevare i dati, in gergo, permette di scegliere la *fonte dei dati*.
- La clausola **WHERE** permette di scegliere i filtri da applicare sui dati. Questa clausola ammette come argomento un'epressione booleana costruita combinando predicati con operatori di **confronto** e operatori **logici**.
- La clausola **GROUP BY**: la clausola **GROUP BY** permette di specificare un raggruppamento rispetto a un particolare attributo
- La clausola **ORDER BY**: la clausola di **ORDER BY** permette di specificare un ordinamento delle righe risultato di un interrogazione rispetto a un particolare attributo. Ad esempio, in una query che ordina rispetto a un'indice numerico progressivo è possibile specificare un ordinamento **crescente** (***ASC***) o **decrescente** (***DESC***).

2 Gestione dei valori nulli

Per poter rappresentare in modo semplice, ma al tempo stesso comodo, la non disponibilità di un valore, il concetto di relazione viene esteso prevedendo anche che una n-upla possa assumere, su ciascun attributo (che non sia chiave primaria), un valore speciale, detto valore ***nullo***, valore che denota appunto la mancanza di un valore.

Per selezionare i termini con valore nullo in SQL è possibile utilizzare il predicato **IS NULL**, la cui sintatti è

```
Attributo IS [ not ] NULL
```

la gestione del valore nullo all'intero di un predicato, secondo lo standard SQL-89, assicura che una qualsiasi operazione logica su un valore **NULL** restituisce **FALSE**.

Se i valori dell'attributo A appartengono al dominio D_A, $t[A]$ è un valore del dominio oppure il valore **nullo**, esistono tre casi in cui l'utilizzo del NULL è coerente:

- Valore **sconosciuto**.
- Valore **inesistente**.
- Valore **senza informazioni**.

L'SQL mette anche a disposizione delle restrizioni sulla possibilità di assegnamento di valore nullo rispetto a un particolare attributo $A \in X$.

3 Gestione dei duplicati

Una differenza significativa tra SQL e l'algebra relazionale riguarda la gestione dei duplicati. In algebra relazionale una tabella viene vista come una relazione matematica e di conseguenza come un'insieme di **tuple** diverse tra di loro. In SQL è invece possibile avere dei duplicati, a patto che i duplicati non riguardino la chiave primaria, in quanto attributo univoco; La rimozione dei duplicati risulta un'operazione estremamente costosa, ciò ha portato alla decisione di includere la keyword DISTINCT, la quale, se specificata successivamente al SELECT, come ultima operazione, rimuove dal **subset** generato dalla query tutti i valori duplicati. La sinstassi è la seguente

```
SELECT [ ALL | DISTINCT ] column_name
FROM table_name
```

l'utente ha quindi la possibilità di scegliere se mantenere i duplicati (opzione di *default*, oppure inserendo la keyword **ALL** dopo il select) oppure di rimuoverli.

4 Gestione delle date

Una data (tipo **DATE()**) è una stringa del formato '**YYYY-MM-DD**', essa rappresenta il tempo trascorso da un **reference**.

Al tipo **DATE()** è possibile applicare la clausola **BETWEEN** o tutti gli operatori di confronto matematici per effettuare ordinamenti. Inoltre, per ottenere un orario, specificando anche l'orario è possibile utilizzare il tipo di dato **TIMESTAMP** che in aggiunta al tipo **DATE** correla annche l'orario nel formato '**HH:MM:SS**'.

Per estrarre facilemente informazioni riguardati un singolo membro di una data si possono utilizzare le funzioni **YEAR()**, **MONTH()** e **DAY()**. In SQL i tipi DATE non si possono né sommare né sottrarre tramite gli operatori matematici + e - , per ottenere la differenza tra due date è necessario utilizzare la funzione **DATEDIFF(Recent, Old)**, funzione che restituisce il numero di giorni che separa le due date specificate.

In SQL esiste una funzione che permette di definire un intervallo esatto in termini di giorni mesi o anni **Data +/- INTERVAL X [YEAR — MONTH — DAY]**, tramite questa funzione è possibile anche sommare e sottrare interi periodi di tempo dalle date, prendiamo ad esempio la query

> *indicare la matricola e il mese di iscrizione degli studenti che si sono laureati dopo 5 anni esatti dal giorno dell'iscrizione*

tramite gli intervalli è possibile semplificare la query nel seguente modo

```
SELECT Matricola
FROM Studenti
WHERE DataLaurea = DataIscrizione + INTERVAL 5 YEAR
```

Inoltre, per sommare e sottrarre date specifiche esistono due funzioni: **DATE_ADD(DATE, INTERVAL)** e **DATE_SUB(DATE, INTERVAL)**.

Per manipolare il formato di una data si utilizza la funzione **DATE_FORMAT(Campo, 'formato')** specificando il formato della data sfruttando alcuni caratteri. Questa operazione permette, tra le altre cose, di facilitare i confronti tra le date, assumiamo di avere la seguente richiesta

> *indicare la matricola degli studenti che si sono laureati di mercoledì*

tramite la funzione **DATE_FORMAT(Campo, 'formato')** possibile definire la query nel seguente modo

```
SELECT Matricola
FROM Studente
WHERE DATE_FORMAT(DataLaurea, '%w') = 3
```

5 Operatori di aggregazione

Gli operatori di aggregazione sono operatori utilizzati per ridurre l'intero **result set** in un unico scalare, talvolta vengono abbinati con la clausola DISTINCT

```
-- Calcola la somma di un insieme di valori
SELECT SUM(column_name) AS total_sum FROM table_name;

-- Calcola la media di un insieme di valori
SELECT AVG(column_name) AS average_value FROM table_name;
```

```sql
-- Calcola il massimo valore di un insieme di valori
SELECT MAX(column_name) AS max_value FROM table_name;

-- Calcola il minimo valore di un insieme di valori
SELECT MIN(column_name) AS min_value FROM table_name;

-- Calcola il numero di righe in un insieme di dati
SELECT COUNT(*) AS row_count FROM table_name;

-- Calcola il numero di righe univocein un insieme di dati
SELECT COUNT(DISTINCT *) AS row_count FROM table_name
```

Questi operatori permettono di eseguire **calcoli** i cui operandi sono i valori assunti da un attributo, in un insieme di record che collassa all'intero di un unico attributo. Prendiamo come esemmpio la seguente query

> *Indicare il numero di visite effettuate in data primo marzo 2013*

la query sarà quindi

```sql
SELECT COUNT(*) AS VisitePrimoMarzo
FROM Visita
WHERE Data = '2013-03-01'
```

6 Join

L'**SQL** mette a disposizione la possibilità di eseguire **query** su più tabelle, queste operazioni risultano particolarmente utili quando si cerca di lavorare su fil

Esistono diverse categorie di **join**:

- **INNER JOIN**: Date due tabelle, combina ogni record della prima con tutti i record della seconda che verificano una determinata.
- **NATURAL JOIN**: Combina una riga della prima tabella con una riga della seconda tabella, se e solo se l'attributo ha lo stesso valore in ambedue i record.
- **OUTER JOIN**:Date due tabelle, combinano ogni record della prima con tutti i record della seconda che soddisfano una condizione, mantenendo tutti i record di una delle due tabelle. Per fare in modo che l'**OUTER JOIN** sia esclusivo e che quindi all'interno del risultato non vi siano i dati relativi all'intersezione è necessario imporre che la chaive primaria della tabella che non viene joinata sia nulla. Esistono due tipologie di **OUTER JOIN**:
  - **LEFT OUTER JOIN**: il join viene fatto rispetto alla relazione presente alla sinistra del **JOIN**

    ```sql
    SELECT * FROM Medico m LEFT JOIN Visite v
    ```

    in questo caso verrà eseguito il **prodotto cartesiano** e verranno estratti nella query di risultato i medici che hanno e non hanno fatto visite.
  - **RIGHT OUTER JOIN**: il join viene fatto rispetto alla relazione presente alla destra del **JOIN**

    ```sql
    SELECT * FROM Medico m RIGHT JOIN Visite v
    ```

    in questo caso verrà eseguito il **prodotto cartesiano** e verranno estratti nella query di risultato le visite che sono state fatte da medici che lavorano nella clinica e da medici che non lavorano più nella clinica.
- **SELF JOIN**: Questa tipologia di JOIN permette di fare il JOIN tra una tabella e se stessa che rispettano una determinata condizione

7 Interrogazioni nidificate

In linguaggio SQL è possibile costruire interrogazioni che contengono al loro interno altre **interrogazioni**, dette **interrogazioni nidificate**. Questa *nidificazione* può avvenire nelle clausole di **SELECT**, **FROM** e **WHERE**, anche se la più comune nidificazione è quella contenuta nella clausola di **WHERE**.

Quando l'interrogazione nidificata compare nel **WHERE**, è possibile confrontarla con un attributo, sfruttando operatori di confronto, oppure *keyword* come **ANY** o **ALL**. In particolare, le due *keyword* specificano:

- **Keyword ANY**: La ***keyword ANY*** specifica che la riga **soddisfa la condizione se risulta vero il confronto tra la riga in esame e almeno una delle righe presenti nel *result-set* della *subquery***.
- **Keyword ALL**: La ***keyword ALL*** specifica che la riga **soddisfa la condizione se risulta vero il confronto tra la riga in esame e tutte le righe presenti nel *result-set* della *subquery***.

Quando viene eseguita un'interrogazione contenente al suo interno al più di una interrogazione nidificata, queste ultime vengono eseguite per prime, il loro **result-set** viene salvato in una **tabella temporanea**, e il controllo sulle righe dell'interrogazione originale può essere fatto accedendo direttamente al risultato temporaneo. Esistono due tipologie principali di interrogazioni nidificate:

- **Interrogazioni nidificate correlate**: in questo tipo di interrogazioni vi è una dipendenza tra la query esterna e la query interna. Questa problematica viene gestita attraverso un passaggio di **binding**.
- **Interrogazioni nidificate non correlate**: in questo tipo di interrogazione non vi è una dipendenza tra l'interrogazione nidificata e la sua query esterna.

L'esistenza delle interrogazioni nidificate correlate porta ad un ulteriore problema: il funzionamento spiegato in precedenza di un'interrogazione nidificata crolla e diventa necessario introdurre una nuova definizione per il funzionamento diquesta tipologia di interrogazione:

> Per ogni riga della **query esterna**, viene valutata subito la **query nidificata**; viene quindi calcolato il predicato a livello di riga sulla **query esterna**.

A **livello di visibilità** rimane la restrizione che *una variabile è usabile solo nell'ambito della query in cui è definita.*

8 Clausola di raggruppamento

Abbiamo caratterizzato gli operatore aggregati come **operatori che vengono applicate ad un insieme di righe**. Talvolta però sorge la necessià di applicare l'operatore aggregato separatamente ad un sottoinsieme di righe, a questo scopo l'SQL implementa un **operatore di raggruppamento**, chiamato **GROUP BY**

```
GROUP BY NomeAttributo
```

Una volta eseguita l'interrogazione, il **GROUP BY** anallizza la tabella dividendo in insiemi caratterizzati da un valore unitario dell'attributo dichiarato nella **clausola di raggruppamento**. Dopo aver completato il raggruppamento in sottinsiemi, per ogni insieme, viene applicato l'operatore aggregato.

La sintassi SQL impone che, in un interrogazione che fa uso della clausola di raggruppamento possa comparire come argomento della SELECT solamente un sottinsieme degli attributi usati nella clausola GROUP BY. Per questi attributi, infatti, ciascna tupla del sottoinsieme sarà caratterizzato dallo stesso valore

8.1 Predicati sui gruppi

L'SQL insieme alla **clausola di raggruppamento** implemente anche un sistema di **predicati sui gruppi**. La clausola **HAVING** descrive le condizioni che si devono applicare al termine dell'esecuzione di un interrogazione che fa uso della clausola GROUP BY. Ogni sottoinsieme di righe che soddisfa il predicato viene inserito all'interno del risultato dell'interrogazione.

9 Interrogazioni insiemistiche

L'SQL mette a disposizione **operatori di tipo insiemistico**, analoghi a quelli definiti nel capitolo sull'algebra relazionale. Gli operatori insiemistici disponibili sono

- **UNION**: Equivalente all'unione logica.
- **EXCEPT**: Equivalente alla sottrazione logica.
- **INTERSECT**: Equivalente all'intersezione logica.

Gli operatori insiemistici, al contrario di ogni altro operatore nel linguaggio, assumono come default l'eliminazione dei duplicati. Qualora però vi sia la necessità di conservare i duplicati è sufficente specificare la keyword ALL. La sintassi degli operatori insiemistici è la seguente

```
SelectSQL {<UNION | INTERSECT | EXCEPT> [ALL] SelectSQL}
```

Inoltre, SQL **non impone che le due relazioni su cui vengono eseguite gli operatori logici**, l'unica condizione richiesta è che vi sia un egual numero di attributi e che i domini degli attributi siano compatibili

10 Operazioni sui dati

La parte del linguaggio SQL nota come DML (Data Manipulation Language) comprende i comandi per modificare e interrogare il contenuto della base di dati. In particolare, per quanto riguarda la modifica esistono tre operazioni principali

Inserimento L'operazione di inserimento consiste nell'aggiunga di un nuovo record all'interno di una tabella SQL. La sintassi generale del comando è

```
INSERT INTO NomeTabella [ListaAttributi]
            <VALUES (ListValori) | SelectSQL>
```

Da questa sintassi è possibile ricavare due forme diverse

- La prima forma permette di inserire singole righe all'interno delle tabelle, specificando come argomento della clausola values l'insieme dei valori che il nuovo record assumerà

  ```
  INSERT INTO Medici(Matricola, Specializzazione) VALUES (4531, "Cardiologia")
  ```

- La seconda forma permette invece di inserire all'interno di una relazione un insieme di attributi filtrati attraverso un interrogazione SQL

  ```
  INSERT INTO MediciMilanesi(
      SELECT * FROM Medico WHERE Citta = "MilanO"
  )
  ```

Cancellazione La cancellazione è un processo che permette di eliminare righe di una base di dati tramite l'operatore **DELETE**.

```
DELETE FROM NomeTabella [WHERE Condizione]
```

Quando la condizione argomento della clausola WHERE **non viene specificata** avviene una cancellazione totale dei record della tabella, qualora invece sia presente, verranno eliminati solo i record che soddisfano la condizione. Si ricorda che nel caso in cui vi sia un vincolo di integrità referenziale con la politica CASCADE la cancellazione di righe da una tabella potrebbe risultare nella cancellazione di record da un'altra.

Chapter 10

Linguaggio SQL, concetti avanzati

1 Vincoli di integrità generica

Il linguaggio SQL permette di specificare un certo insieme di vincoli sugli attributi e sulle tabelle, soddisfacendo le esigenze di verifica dell'integrità di alcuni dati. Per specificare questi vincoli SQL mette a disposizione una sintassi

```
check (condizione)
```

Le condizioni ammissibili sono le stesse che possono apparire come argomento della clausola where di un'interrogazione SQL. La condizione deve essere verificata affinché la base di dati sia corretta.

Questi vincoli trovano un enorme applicabilità nel linguaggio SQL, una dimostrazione la si ha nella possibilità di rappresentare tutti i vincoli predefiniti attraverso questa clausola. Per esempio, si assuma di voler creare la tabella impiegato

```
CREATE TABLE Impiegato (
    Matricola varchar(5)
            CHECK (Matricola IS NOT NULL AND (SELECT COUNT(*)
                                              FROM Impiegato I
                                              WHERE Matricola = I.Matricola) = 1)
    Cognome varchar(30) CHECK (Cognome IS NOT NULL)
    Nome varchar(30) CHECK (Cognome IS NOT NULL)
)
```

In questo particolare caso è evidente che le condizioni espresse con il check sono esprimibili, in modo maggiormente efficiente, tramite altri vincoli. La vera potenza del vincolo generico sta, in questo caso, nella possibilità di poter aggiungere anche vincoli che impongano la necessità di avere un superiore per ogni dipendente

```
Superiore varchar(7),
check(Matricola like '1%' or Dipart = (select Dipart
                                       from Impiegato I
                                       where I.matricola = Superiore)
```

2 Asserzioni

Grazie alla clausola **CHECK** è possibile definire anche un ulteriore componente dello schema della base di dati, le ***asserzioni***. In SQL queste asserzioni sono dei **vincoli** che **non sono associati a un attributo o a una tabella**, ma appartengono direttamente allo schema. Mediante le asserzioni è possibile esprimere tutti i vincoli che abbiamo specificato nelle definizioni di tabelle, inoltre, le asserzioni permettono di esprimere vincoli che non sarebbero altrimenti esprimibili, quindi, vincoli che coinvolgono più tabelle o che richiedano che una tabella abbia una cardinalità minima.

La sintassi per creare un **asserzione** è la seguente

```
CREATE ASSERTION NomeAsserzione CHECK (condizione)
```

Un asserzione potrebbe ad esempio imporre che vi sia almeno uno studente per ogni classe di un istituto scolastico

```
CREATE ASSERTION AlmenoUnoStudente CHECK (
                        SELECT count(*)
                        FROM Studente
                        WHERE Classe = "C1" > 1)
```

Ad ogni vincolo di integrità, definito tramite **check**, è associata una una politica di controllo che specifica se il **vincolo è immediato** o se il **vincolo è differito**. Nel primo caso i vincoli sono verificati *ad ogni modifica della base di dati*, mentre i **vincoli differiti** vengono controllati solo al termine dell'esecuzione di una serie di operazioni.

Il controllo differito viene utilizzato per gestire casi in cui non è possibile costruire uno stato consistente della basse di dati con una singola modifica. L'esecuzione di un comando che modifica la base di dati, se rispettati vincoli e condizioni, restituisce a sua volta una base di dati che rispetta tali vincoli.

3 Viste

Nei paragrafi precedenti è stato introdotto il concetto di **vista**, rappresentabile, in linea di massima, come una tabella *virtuale* o *fisica* il cui contenuto dipende dal contenuto di altre tabelle.

In SQL le **viste** vengono definite in SQL associando un nome e una lista di attributi al risultato dell'esecuzione di un interrogazione. A livello di linguaggio, per definire la vista, si utilizza il comando

```
CREATE VIEW NomeVista [(ListaAttributi)]
AS SelectSQL
[WITH [LOCAL | CASCADED] CHECK OPTION]
```

L'interrogazione SQL (**SelectSQL**) deve restituire un insieme di attributi compatibile con gli attributi nello schema della visita: l'ordine nella clausola select deve infatti corrispondere a quello in **ListaAttributi**. Ad esempio, si assuma di voler creare una vista ***medici_poveri*** che includa i medici on una parcella inferiore a 100 euro

```
CREATE VIEW medici_poveri (Matricola, Nome, Cognome, Parcella) AS (
    SELECT *
    FROM Medico
    WHERE Parcella < 100
)
```

Su alcune viste è possibile effettuare anche delle operazioni di modifiche, che verranno tradotte negli opportuni comandi a livello delle tabelle di base. Non è però sempre possibile determinare un modo univoco in cui la modifica sulla vista possa essere riportata sulle tabelle di base; si incontrano problemi sopratutto quando la vista è definita tramite un **join** tra più tabelle. Lo standard SQL permette che una vista sia aggiornabile quando una sola riga di ciascuna tabella di base corrisponde a una sola riga della vista.

Nonostante ciò i sistemi commerciali permettono che una vista sia aggiornabile solo quando essa è derivata da una sola tabella. In questi casi ristretti la clausola **CHECK OPTION** può essere usata e specifica che a seguito della modifica ogni riga della vista sia comunque appartenente alla vista.

In SQL esiste però un meccanismo per aggiornare le visite, chiamato anche **refresh**. Una vista materializzata può essere aggiornata in due modi

- **Full Refresh**: aggiorna la vista partendo completamente da 0. Esistono tre principali politiche di full refresh: **immediate refresh**, **deferred refresh** e **on deman refresh**. La **deferred refresh** può essere implementata tramite **trigger**, la **on demand refresh** può essere fatta utilizzando una **procedura** e la **deferred refresh** può essere fatta utilizzando un **evento**.

- **Incremental Refresh**: aggiorna solo la parte della vista non più aggiornata. Esistono tre tipologie di **incremental refresh**:
  - **partial refresh**: trasferimento parziale del log.
  - **complete refresh**: trasferimento completo del log.
  - **rebuild**: ricalcolo completo, nella pratica equivale a una **full refresh**

  in generale, l'**incremental refresh** è migliore poiché è possibile aggiornare la vista materializzata sfruttando i dati che essa già contiene e la log table,

Abbiamo introdotto nel paragrafo la **log table** senza specificare cosa essa sia. La **log table** o **tabella di log** è una tabella il cui scopo è **tenere traccia di tutte le modifiche delle tabelle del database sulle quali si basano i suoi dati**. Quindi, quando si decide di effettuare il refresh, i dati nella **log table** devono essere sufficienti per effettuarlo. Inoltre, se la vista si basa su più tabelle, potrebbe essere utile avere più **log table**.

4 Procedure

L'SQL mette a disposizione la possibilità di implementare delle ***stored-procedure***, queste procedure hanno come scopo principale quello di memorizzare all'interno della base di dati delle operazioni **standard**, quindi operazioni che possono essere richiamati in tempi diversi, da utenti diversi e con una certa frequenza; difatti quando una procedura viene definita essa viene registrata al pari di qualsiasi funzione SQL. La sintassi delle procedure è

```
CREATE PROCEDURE StudentiDiUnaClasse(:Classe varchar(3))
BEGIN
    SELECT *
    FROM Studente
    WHERE classe = Classe
END;
```

Le procedure sono un ottimo sistema per risparmiare tempo utile alla scrittura dei comandi. Inoltre, esse permettono anche di svolgere una richiesta **condizionata**, tramite l'utilizzo del costrutto **if-else**. Per fare un esempio, si assuma di voler creare una procedura che aggiunga uno studente ad una classe se essa non è già piena

```
CREARE PROCEDURE InterrogazioneDiRecupero(:MatricolaNuovoStudente varchar(8),
                                          :NuovaClasse varchar(3))
BEGIN
    IF (
        SELECT COUNT(*)
        FROM Studenti s INNER JOIN Classe c
                ON s.Classe = c.Codice
        WHERE c.Classe = NuovaClasse
        GROUP BY c.Classe

    ) < 21
        INSERT INTO Studente (Matricola) VALUES (MatricolaNuovoStudente)
```

Lo standard di SQL-2 non tratta la scrittura di procedure SQL complesse, limitandosi a specificare soltanto la definizione di procedure composte da un singolo comando SQL. Molti sistemi rimuovono questa limitazione, andando incontro alle esigenze delle applicazioni, attraverso delle estensioni procedurali, le quali permettono per l'appunto di rimuovere queste limitazioni introducendo altri elementi (ad esempio, il costrutto if-else).

Come è anche stato accennato all'inizio del paragrafo, la chiamata ad una procedura SQL può essere fatta nel seguente modo

```
CALL NomeProcedura();
```

Attraverso le procedure è possibile anche implementare **soluzioni iterative**, le quali sfruttano i costrutti classici della programmazione: **while**, **for**, **loop** e . Ad esempio, la struttura generale di un **while** può essere espressa come

```
WHILE condition DO
    -- Bloco di istruzioni
END
```

Quando si utilizza questo ciclo la condizione *viene valutata prima dell'esecuzione del codice*. Se l'obiettivo è invece fare l'opposto, è possibile utilizzare il costrutto **repeat-untill**, strutturato nel seguente modo

```
REPEAT
    -- Blocco di istruzioni
UNTIL condition
END REPEAT;
```

Ad esempio, si assuma di voler scrivere una **stored-procedure** che: *ricevuto in ingresso un intero i, stampi a video i primi i interi, separati da virgola e in ordine crescente*

```
CREATE PROCEDURE stampa_primi_n_interi (IN _i INT)
    BEGIN
        DECLARE count INT DEFAULT 0;
        DECLARE string VARCHAR(255);
        WHILE (count <= i)
            SET s = CONCAT(s, ' ', counter);
            SET count = count + 1;
        END WHILE;

        SELECT S;
    END
```

l'ultima opzione resa disponibile dall'SQL è quella di utilizzare una **loop**, la quale sintatticamente si definisce come

```
loop_label: LOOP
    -- blocco di istruzioni e check di condizioni
END LOOP;
```

le condizioni di uscita sono gestite dal programmatore stesso. Insieme ai **loop** diventa fondamentale l'utilizzo delle **istruzioni di salto**. In SQL esistono due **istruzioni di salto**:

- **Istruzione *LEAVE***: permette di interrompere un ciclo.
- **Istruzione *ITERATE***: permette di passare all'iterazione successiva.

Ad esempio, si assuma di voler riprodurre la query precedente utilizzando il **loop statement**.

```
CREATE PROCEDURE stampa_primi_n_interi (IN _i INT)
    BEGIN
        DECLARE count INT DEFAULT 0;
        DECLARE string VARCHAR(255);

        scan: LOOP
            SET count = count+1;

            IF count = i+1 THEN
                LEAVE scan;
            END IF
            IF count%2 == 0 THEN
                ITERATE scan;
```

```
            ELSE
                SET s = CONCAT(s, ',', counter);
            END IF;
        END LOOP;
        SELECT s;
    END
```

4.1 Variabili locali per memorizzazione intermedia

Può capitare talvolta che, durante la creazione di una procedura sia necessario dover **memorizzare temporaneamente delle informazioni**, in SQL ciò viene reso possibile attraverso l'ausilio di **variabili locali**. In generale, queste variabili devono essere dichiarate all'inizio del corpo della procedura, con la seguente sintassi

```
DECLARE nome_variabile type(size) DEFAULT valore_default
```

Quando è stata dichiarata opportunamente la variabile è possibile assegnarle un valore in due modalità differenti, la prima, tramite l'istruzione **SET**

```
SET nome_variabile = valore_dominio
```

Il secondo modo, invece, prevede l'utilizzo di una combinazione di **SELECT** e **INTO**, ad esempio

```
SELECT count(*) INTO nome_variabile
```

Una cosa che è importante notare è la seguente

> Una **Variabile non può** contenere un **result-set**.

Esistono infine delle variabili **user-defined**, le quali sono inizializzate dall'utente **senza necessità di dichiarazione**, e il loro ciclo di vita equivale alla durata della connessione a MySQL.

4.2 Parametri di una stored procedure

Una **stored-procedure** in MySQL accetta parametri di vario tipo: **ingresso**, **uscita** e **ingresso-uscita**. Quando all'interno di una procedura viene passato un parametro, *esso potrà essere letto, ma non modificato*. Ad esempio, si assuma di voler creare una procedura che trovi tutti i medici per specializzazione

```
CREATE PROCEDURE CercaMediciPerSpecializzazione(IN _specializzazione: varchar(100))
    BEGIN
        SELECT *
        FROM Medico
        WHERE Specializzazione = _specializzazione;
    END
```

Un parametro può essere modificato per assumere il valore del risultato della stored-procedure; Le variabili utilizzate prendono il nome di **user-defined**. Ad esempio, si assuma di voler cercare tutti i medici per una specifica specializzazione, definendone anche il numero.

```
CREATE PROCEDURE CercaMediciPerSpecializzazione(IN _specializzazione: VARCHAR(100),
                                                OUT totale_medici_ INT)
    BEGIN
        SELECT COUNT(*) INTO totale_medici_
        FROM Medico
        WHERE Specializzazione = _specializzazione;
    END
```

5 Cursori e Fetch

Un **cursore** in SQL adempie a compiti molto simili a quelli di un **puntatore** in un qualsiasi altro linguaggio di programmazione. Per definizione, un cursore

> Scorre i **record** di un **result-set**

Se assumiamo che il result-set sia analogo a una **lista**, possiamo facilmente arrivare alla conclusione che lo scorrimento può avvenire solo "*in avanti*". Per dichiarare un cursore di utilizza la seguente sintassi

```
DECLARE nome_cursore CURSOR FOR SQL_query
```

I cursori si possono dichiarare solo **successivamente alla dichiarazione delle variabili**. La fase di vita di un cursore di articola in tre fasi

- **open**: "apertura" del **cursore**. La sintassi di questa operazione è

  ```
  OPEN nome_cursore;
  ```

- **fetch**: prelievo dei dati riga per riga. La sintassi di questa operazione è

  ```
  FETCH nome_cursore INTO ListaVariabili;
  ```

- **close**: "chiusura" del **cursore**. La sintassi di questa operazione è

  ```
  CLOSE nome_cursore;
  ```

In SQL esistono anche degli **handler**, la cui traduzione letterale è *gestori*, questi costrutti sono infatti dei veri e propri **gestori di situazioni**, utili per riconoscere la fine del **result-set**. Possono essere definiti a seguito della dichiarazione delle variabili e dei cursori, la loro sintassi è la seguente

```
DELCARE CONTINUE HANDLER FOR NOT FOUND
SET finito = 1;
```

Questo **handler** si attiva quando, durante lo scorrimento di un result-set da parte di un cursore, si arriva alla fine di esso e non vengono trovati ulteriori elementi. Ad esempio, assumiamo di voler creare una procedura che prenda tutti i medici per una specializzazione e conta quelli che si chiamano con un certo nome;

```
CREATE PROCEDURE CercaMediciPerSpecializzazione(IN _specializzazione: varchar(100),
                                IN _nome varchar(255), OUT numero_medici)
    BEGIN
        DECLARE finito INTEGER DEFAULT 0;

        DECLARE cur FOR (
            SELECT *
            FROM Medico
            WHERE Specializzazione = _specializzazione;
        )

        DECLARE CONTINUE HANDLER FOR NOT FOUND
        SET finito = 1;

        scan: LOOP
            FETCH cur
                -- controllo sui dati
            IF finito == 0
                -- processo il record;
            ELSE
                LEAVE scan;
            END LOOP scan;
    END
```

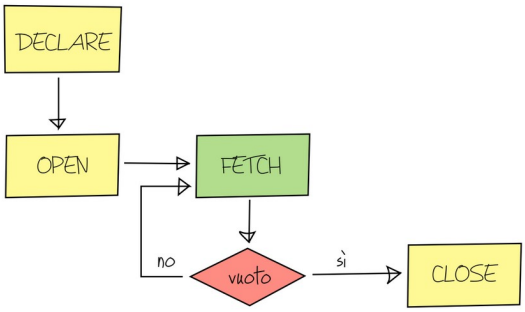


Figure 10.1: Funzionamento di un cursore SQL

Il funzionamento di un cursore può essere schematizzato nel seguente modo
Facciamo un ulteriore esempio, si assuma di voler scrivere una **stored-procedure** che riceva in ingresso una specializzazione s e restituisca i codici fiscali dei pazienti visitati da un solo medico di s, in una stinga del tipo "codice fiscale 1, codice fiscale 2, ..."

```
CREATE PROCEDURE pazienti_visitati(IN specializzazione_ varchar(255),
                                   OUT _codici_fiscali TEXT)
    BEGIN
        -- Dichiarazione delle variabili
        DELCARE finito INT DEFAULT 0;
        DECLARE codice_fiscale VARCHAR(255) DEFAULT ' ';

        -- Dichiarazione del cursore
        DECLARE codici_fiscali CURSOR FOR (
            SELECT v.CodFiscale
            FROM Medico m INNER JOIN Visite v
                    ON m.Matricola = v.Visita
            WHERE m.Specializzazioe = _specializzazione
            GROUP BY v.CodFiscale HAVING COUNT(DISTINCT v.Medico) = 1;
        )

        -- Dichiarazione dell'handler
        DECLARE CONTINUE HANDLER FOR NOT FOUND
        SET finito = 1;

        -- Apertura del cursore
        OPEN medici

        scan: LOOP
            FETCH cur INTO codice_fiscale
            IF finito == 1 THEN
                LEAVE preleva;
            END IF;
            SET _codici_fiscali = CONCAT(_codici_fiscali, ' ', codice_fiscale);
        END LOOP scan;
        -- Chiusura del cursore
        CLOSE medici
    END
```

una volta definita la procedura essa può essere richiamata facendo

```
SET @codici_pazienti = '';
CALL pazienti_visitati('Ortopedia', @codici_pazienti);
```

```
SELECT @codici_pazienti;
```

6 Trigger

SQL fornisce un costrutto estremamente utile e potente: il **trigger**. Questo strumento rende la base di dati in grado di reagire a eventi definiti dall'amministratore tramite l'esecuzione di opportune azioni. Una base di dati dotata di tale capacità viene definita **attiva**, essa dispone di un *sottosistema integrato per definire e gestire regole di produzione (**regole attive**).*

Le **regole attive** seguono il *paradigma di Evento-Condizione-Azione*: ciascuna **regola** reagisce ad alcuni eventi, valuta una condizione e in base al **valore di verità** della condizione, esegue o meno una **reazione**. L'esecuzione delle regole avviene sotto il controllo di un *sistema autonomo*, detto **processore delle regole** (*rule engine*), il quale tiene traccia degli eventi e manda in esecuzione le regole in base a proprie politiche; il sistema risultante ha un comportamento detto **reattivo**.

Quando una base di dati ha un **comportamento reattivo**, una parte dell'applicazione normalmente codificata mediante i programmi può essere *espressa mediante delle regole attive.* Queste regole possono gestire diversi aspetti di una base di dati: **vincoli di integrità**, *calcolare dati derivati*, ecc... Questo strumento permette quindi di aggiungere un'ulteriore caratteristica di dati, la quale si affianca alla già presente **indipendenza delle base di dati**: ***l'indipendenza della conoscenza***.

La creazione di un **trigger** appartiene al **DDL** (*Data Definition Language*); i **trigger** possono anche essere rimossi e, in alcuni sistemi, attivati e disattivati all'occorrenza. Come anticipato precedentemente, i **trigger** seguono il paradigma di **Evento-Condizione-Azione**:

- Gli eventi sono primitive per la manipolazione dei dati: **INSERT**, **UPDATE** e **DELETE**.
- La condizione è un **predicato logico booleano**, espresso in SQL.
- L'azione è una sequenza di primitive SQL generiche, talvolta arricchite da un linguaggio di programmazione integrato.

In genere i trigger fanno riferimento a una tabella detta **target**.

Il paradigma **ECA** ha un comportamento intuitivo:

> *Quando* si verfica l'evento *se* la condizione è soddisfatta, *allora* viene svolta l'azione.

Quando ciò avviene si dice che *un **trigger** è stato attivato da uno dei suoi eventi, viene **valutato** durante la verifica della sua condizione e viene **eseguito** quando la valutazione è positiva.* I **trigger** hanno due diversi livelli di attivazione

- **trigger di tupla (***row-level***)**: l'attivazione avviene per ogni tupla coinvolta nell'operazione; Si ha quindi un comportamento **orientato alle singole istanze**.
- **trigger di primitiva(***statement-level***)**: l'attivazione avviene una volta sola per ogni **primitiva SQL** facendo riferimento a tutte le tuple coinvolte nelal primitiva; Si ha quindi un comportamento orientato agli insiemi.

Vi è un'ulteriore categorizzazione dei trigger, quelli che lavorano in modo *immediata*, la cui valutazione avviene o immediatamente dopo l'evento che li ha attivati o, più raramente, precedentemente ad esso, oppure, ci sono i **trigger** che lavorano in modo **differito**, quindi a seguito della transazione (terminata da un comando **COMMIT WORK**).

La sintassi generale di un trigger SQL si articola nel seguente modo

```
CREATE TRIGGER nome_trigger
    modo evento ON tabella_target
    [REFERENCING referenza]
    [FOR EACH livello]
    [WHEN (predicato_sql)]
    statement_procedurale_sql
```

dove il *modo evento* specifica se l'operazione deve essere fatta prima o dopo la fine dell'evento, l'evento è l'operazione da attuare (**INSERT**, **DELETE** e **UPDATE**), il *livello* permette di specificare se l'operazione è *row-level* o *statement-level*. Le clausole opzionali possono specificare

- *referenza*: consente di introdurre dei nomi di variabili rispetto ai default. La sintassi sfrutta i costrutti **OLD** e **NEW**

```
OLD AS VariabileOld|NEW AS VariabileNew
```

7 Eventi

Gli **eventi** sono, per definizione, *stored programs eseguiti in base a condizioni* ***dettate dal tempo***. In generale, la sintassi di un evento è

```
CREATE [DEFINER = user] EVENT [IF NOT EXISTS] event_name
ON SCHEDULE schedule
[ON COMPLETION [NOT] PRESERVE]
[ENABLE | DISABLE | DISABLE ON SLAVE]
[COMMENT 'string']
DO event_body;
```

l'**event** è un costrutto estremamente potente, lo si può dimostrare con un esempio. Si assuma di avere una base di dati strutturata per gestire un ospedale e si assuma di voler

> creare e mantenere giornalmente aggiornata una ridondanza nella tabella Medico, contenente, per ciascuno, il totale di visite effettuate

La quale può essere facilmente risolta impostando un evento

```
CREATE EVENT [IF NOT EXISTS] Aggiorna_visite_medico
ON SCHEDULE EVERY 1 DAY -- Periodicità
STARTS '2024-05-09 00:00:00' -- Prima esecuzione
DO (
    UPDATE Medico
    SET TotaleVisite = TotaleVisite + (
                SELECT COUNT(*) AS visite_effettuate
                FROM Medico m INNER JOIN Visite v
                        ON m.Matricola = v.Medico
                WHERE v.Data = CURRENT_DATE
                )
    )
```

Esiste anche la possibilità di creare eventi che hanno un **inizio** e una **fine** o eventi a **singolo scatto**.

Per poter attivare la schedulazione degli eventi è necessario cambiare il valore della variabile **event_scheduler** a ON

```
SET GLOBAL event_scheduler = ON
```

8 Controllo degli accessi

La presenza di **meccanismi di protezione** dei dati *riveste enorme importanza all'interno di una base di dati*. Il compito più importante di un amministratore di una base di dati consiste nell'**implementare politiche di controllo e accesso alla base di dati stessa**. SQL stesso riconosce l'importanza di queste implementazioni e mette a disposizione dei comandi per poterle sviluppare. In particolare, SQL **prevede che ogni utente che accede al sistema sia riconosciuto in modo univoco** dal sistema.

Il sistema tramite queste politiche protegge **tabelle** e **viste** della base di dati, con la possibilità di specificare singoli attributi all'interno di esse, proteggendo anche elementi come **domini**,

procedure, ecc... Di regola, un utente che crea una risorsa è autorizzato a praticare qualsiasi operazione su di essa, ma l'SQL sarebbe estremamente limitato se fosse possibile solo per l'utente creatore accedervi o sarebbe estremamente insicuro se chiunque potesse accedervi. A tale scopo, l'SQL mette a disposizione dei meccanismi di **privilegio**. Ogni privilegio è caratterizzato da

- La risorsa a cui si riferisce.
- L'utente che concede il privilegio.
- L'utente che riceve il privilegio.
- L'azione che viene permessa sulla risorsa.
- Possibilità di trasmettere a terzi il privilegio.

Quando una risorsa viene creata, il **sistema concede al suo creatore tutti i privilegi su di essa**. Esiste inoltre un utente di default, chiamato _system, il quale rappresenta il **DBA** (*Database Administrator*).

I privilegi definiti di default in SQl sono

- **INSERT**: permette l'inserimento di un nuovo oggetto nella risorsa.
- **UPDATE**: permette di aggiornare un oggetto all'interno della risorsa.
- **DELETE**: permette di eliminare un oggetto dalla risorsa.
- **SELECT**: permette la lettura della risorsa.
- **REFERENCES**:permette che venga fatto un riferimento a una risorsa nell'ambito della definizione dello schema di una tabella.
- **USAGE**: permette l'utilizzo della risorsa.

Il privilegio di poter eseguire un **DROP** o un **ALTER** rimane di esclusiva competenza del creatore della risorsa. In SQL per concedere un privilegio si utilizza il comando **grant**

```
GRANT Privilegi ON Risorsa TO Utenti [WITH GRANT OPTION]
```

Il comando permette di concedere *Privilegi* su una particolare *Risorsa* all'*Utente*. Se invece l'obiettivo è quello di revocare un particolare privilegio è necessario applicare il comando **revoke**, la cui sintassi è la seguente

```
REVOKE Privilegi ON Risorsa TO Utenti [ RESTRICT | CASCADE ]
```

L'unico utente che può rimuovere un privilegio è, ovviamente, quello che lo ha concesso. Le due opzioni vengono utilizzate nel caso in cui la revoca di un privilegio vada a colpire anche altri utenti, in particolare:

- se l'opzione scelta è **restrict** e se la revoca va a colpire anche altri utenti, non viene eseguita.
- se l'opzione scelta è **cascade**, l'eliminazione avviene in ogni caso.

Per semplificare l'utilizzo dei privilegi, l'SQL ha introdotto un modello di controllo degli accessi basato sul modello **role-based-control**. Tramite la creazione di ruoli, i quali non sono altro che dei grandi **contenitori di privilegi**, è possibile assegnare dei gruppi di privilegi a utenti del sistema senza aver bisogno di specificare ogni volta quali privilegi voglio assegnare.

```
CREATE ROLE NomeRuolo
```

Successivamente, per assegnare dei privilegi, è sufficiente utilizzare i comandi descritti sopra, riferendosi, invece che all'utente al ruolo.

9 Window Function

Sono delle funzioni che affiancano a ogni record r un valore ottenuto da un'operazione eseguita su un insieme di record logicamente connessi a r. Ad esempio, si assuma di voler creare la seguente query

> Scrivere una query che indichi, per ogni cardiologo, la matricola, la parcella e la parcella media della sua specializzazione

Per farla è necessario usare la clausola **OVER()**, la quale applica una funzione di tipo **aggregate** o **non-aggregate** a un **insieme di record associati a un record (*partion*) di un result-set**.

```
SELECT
    m.Matricola
    m.Parcella,
    AVG(m.Parcella) OVER()
FROM Medico m
WHERE m.Specializzazione = "Cardiologia"
```

se **over** non ha alcun parametro, la partion utilizzata è sempre la stessa per ogni record e coincide con il result-set. Possiamo definire come partition *insieme di record a cui si applica una funzione aggregate o non-aggregate e* ***dipende dal record processato***. Prendiamo come esempio un'altra query

> Scrivere una query che indichi, per ogni medico, la matricola, la specializzazione, la parcella e la parcella media della sua specializzazione

La quale può essere svolta attraverso una **window function**

```
SELECT
    m.Matricola,
    m.Specializzazione,
    m.Parcella,
    AVG(m.Parcella) OVER(
                    PARTITION BY
                    m.Specializzazione)
FROM Medico m;
```

l'istruzione **AVG** viene applicata partizionando i record del result set della query per specializzazione; ad ogni record del result-set si associa poi il valore AVG relativo alla corrispondente specializzazione. Con le **window-function** è possibile utilizzare anche delle funzioni **non aggregate**, quest'ultime invece che lavorare su **partition** lavorano su **frame** e inoltre

> Associano a ciascun record di un result-set un valore ottenuto dalla partition senza fondere i record in un'informazione riepilogativa.

Un esempio è la funzione **ROW_NUMBER**, la quale associa un **numero** ad ogni riga del result-set (specificando qualche clausola se necessario). Prendiamo come esempio la seguente query

```
SELECT
    m.Matricola,
    m.Cognome,
    m.Specializzazione,
    ROW_NUMBER() OVER(PARTITION BY m.Specializzazione)
FROM
    Medico m;
```

Il result-set, per ogni possibile insieme ottenuto considerando le specializzazioni, assegnerà a ogni elemento del sottoinsieme un numero di riga. Un altro esempio che lavora su **frame** è la funzione **RANK()**, la quale permette di stilare una classifica *dipendentemente da un criterio*. Prendiamo come esempio la query

> Effettuare una classifica dei medici di ogni specializzazione dipendentemente dalla loro parcella, partendo dalla più alta. Restituire matricola, cognome, specializzazione, parcella e posizione in classifica

Quello che verrà restituito è una classifica, ottenuta prendendo il medico alla **CURRENT_ROW**, prendendo il sottoinsieme di medico della rispettiva specializzazione e numerando le righe in funzione di un criterio, che nel nostro caso è la parcella. Il risultato sarà quindi la classifica dei medici più economici per ogni specializzazione

```
SELECT
    m.Nome,
    m.Congome,
    m.Specializzazione,
    RANK() OVER w
FROM Medici;

WINDOW w AS (PARTITION BY m.Specializzazione
             ORDER BY m.Parcella DESC)
```

Analogamente alla funzione **RANK()** esiste anche la funzione **DENSE_RANK**, la quale, in caso si verifichino degli *ex-aequo* tra due elementi, non salta il valore in classifica.

Altre due funzioni estremamente utili, sono la LEAD e la LAG. Queste due funzioni *ricavano il valore di un attributo di una row posta n posizioni prima, nel caso di **LAG**, o n posizioni dopo, nel caso di **LEAD**.*

9.1 Window function su frame

Questo tipo di window function lavorano sui frame

> Un **frame** è un **sottoinsieme della partition dipendente dalla current row**.

Processano un result set e calcolano valori sulla base di record **adiacenti** alle current row. Tutte le funzioni aggregate, se utilizzare su frame, calcolano il risultato usando solo i record dei frame. Due funzioni che possono essere utilizzate su frame sono le funzioni **LAST_VALUE** e **FIRS_VALUE**.

Una tecnica interessante è quella del **moving avarage**. Per spiegarla è meglio procedere con un esempio

> Scrivere una funzione analytics che, per ogni terapia conclusa del paziente 'ttw2', restituisca il farmaco, la durata e la durata media rispetto alla terapia precedente e successiva con lo stesso farmaco

La cui soluzione sarebbe

```
WITH durate AS (
    SELECT
        T.Farmaco,
        T.DataInizioTerapia,
        DATEDIFF(
            T.DataFineTerapia,
            T.DataInizioTerapia
        ) AS Durata
    FROM Terapia T
    WHERE T.Paziente = "ttw2"
            AND T.DataFineTerapia IS NOT NULL
)
SELECT
    D.Farmaco,
    D.Durata,
    D.DataInizioTerapia,
```

```
    AVG(D.Durata) OVER w
FROM durate D;

WINDOW w AS (ORDER BY D.DataInizioTerapia
    ROWS BETWEEN 1 PRECIDING AND 1 FOLLOWING)
```

Nella prima parte definiamo **durate**, quindi l'insieme delle terapie **concluse** del paziente con codice fiscale "ttw2" e ne prendiamo il *Farmaco*, la *data inizio terapia* e la sua durata. Nella seconda parte invece andiamo a prelevare gli stessi parametri, aggiungendo però la window function, la quale calcola la durata media della terapia, riguardante sempre "ttw2", ma sulla terapia immediatamente precedente, sulla terapia immediatamente successiva e su quella della current row.

La fondamentale utilità delle window function risiede nella loro possibilità di muoversi all'interno del result-set liberamente.

Part V

Appendici

Appendix A

Appendice 0: Nomenclatura di una tabella

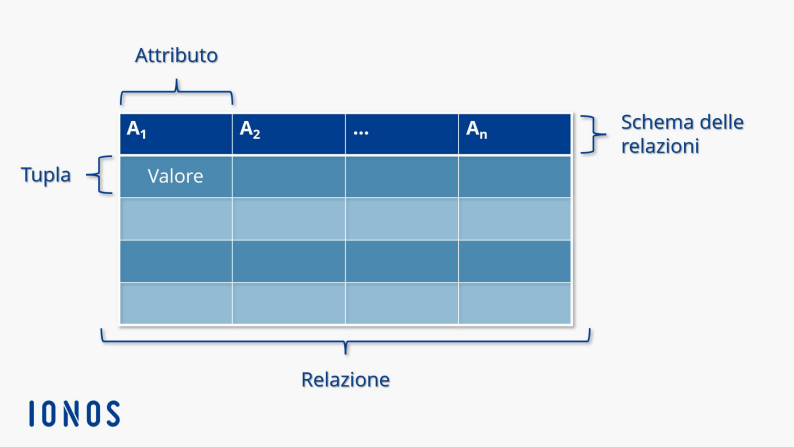


Figure A.1: Struttura di una relazione

Appendix B

Appendice 2: Come leggere un diagramma E-R

La lettura di un diagramma E-R è un procedimento che potrebbe diventare estremamente ostico per alcuni, questo paragrafo si prefigge di definire uno standard, seguito anche dal professore per la lettura del diagrammi. Si assuma quindi di star modellando la seguente realtà Per leggerlo è

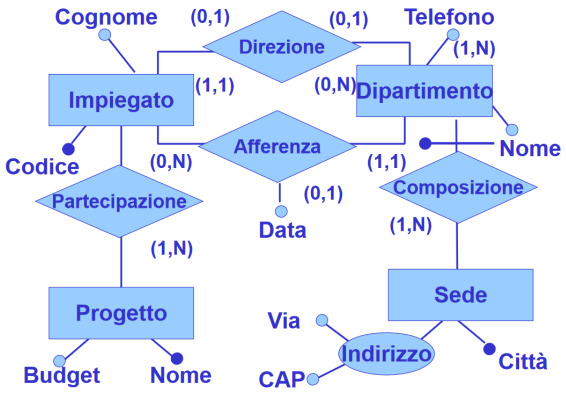


Figure B.1: Schema E-R

necessario riferirsi alle singole relazioni

- **Relazione Sede-Dipartimento**
  - Da **(1,n)** dipartimenti compongono **una e una** sola sede.
  - **Una e una** sola sede è composta da **(1,n)** dipartimenti.
- **Relazione Impiegato-Dipartimento (Afferisce)**:
  - Da **(0,n)** impiegati afferiscono ad **uno e un solo** dipartimento.
  - Ad **uno e un solo** dipartimento afferiscono **(0,n)** impiegati
- **Relazione Impiegato-Progetto**:
  - Da **(1,n)** impiegati partecipano a **(0,n)** progetti (quindi, quando vi è un progetto, deve partecipare da un minimo a 1, fino a n impiegati, ma non è detto che questi impiegati debbano per forza partecipare al progetto).
  - In **(0,n)** progetti partecipano **(1,n)** impiegati.
- **Relazione Impiegato-Dipartimento (Dirige)**:

– Un impiegato può dirigere un dipartimento.
– Un dipartimento può essere diretto da uno e un solo impiegato.

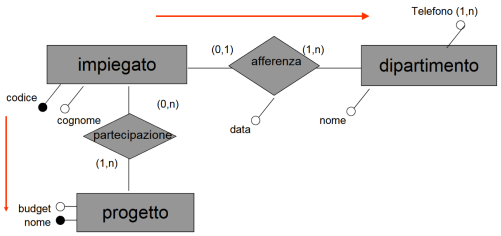
Appendix C

Approfondimento 1: Calcolo delle tavole di accesso

Per calcolare una tavola degli accessi è necessario innanzitutto avere a disposizione una **tabella dei volumi**. Quindi una tabella all'interno della quale, per ogni elemento, del nostro diagramma E-R (**entità** o **relazione**) venga specificato il **volume indicativo** o, in altri termini, il **numero dei record**. Dopo aver definito la tabella dei volumi è necessario stabilire per quale operazione vogliamo calcolare la **tavola degli accessi** e fissare uno **schema di navigazione**, quindi, quella porzione di diagramma E-R necessaria allo svolgimento dell'operazione.

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

(a) Tabella dei volumi



(b) Descrizione dell'immagine 2

Figure C.1: Descrizione generale delle due immagini

La tabella degli accessi riporta, per ogni entità, oltre al suo tipo, anche il **numero di accessi**, il **numero di istanze per accesso** e il **tipo degli accessi** che vengono fatti di su di essa: **accesso in lettura**, **accesso in scrittura**. Il numero di istanze può essere ricavato mediante semplici operazioni sulla tabella dei volumi. Ad esempio, assumiamo di voler calcolare la tavola degli accessi per la seguente operazione

> Estrapolare tutti i dati di ogni dipendente, in più il dipartimento dove lavora e i progetti in cui è coinvolto

I dati dei dipendenti richiedono un solo accesso in lettura alla rispettiva tabella. Per estrapolare i dati del dipartimento dobbiamo analizzare invece la cardinalità della relazione, la quale è $(1-n)$, quindi sappiamo che *un singolo impiegato* afferisce a un solo dipartimento (il fatto che più impiegati possano afferire allo stesso dipartimento è ininfluente per la nostra operazione), di conseguenza abbiamo un solo accesso alla tabella afferenza (che ci dice a quale dipartimento afferisce il dipendente) e, pertanto, un solo accesso alla tabella dipartimento. Per i progetti la questione è diversa, la relazione è infatti di tipo $(n-n)$, dobbiamo stimare per ogni impiegato a quanti progetti di media partecipa, all'interno della tabella dei volumi abbiamo infatti 6000 progetti e 2000 dipendenti, facendo un semplice rapporto abbiamo che **ogni dipentente partecipa in media a tre progetti**, quindi abbiamo tre accessi alla relazione partecipa e tre accessi alla relazione progetto.

Bibliography

[1] P.Azteni, S. Ceri (2023) *Basi di dati VI Edizione*, Mc Graw Hill.

[2] Slide Prof. Pistolesi (2024)

[3] Slide Prof. Tonellotto (2024)

[4] Documentazione ufficiale MySql (2024)