

Basi di dati

VR443470

maggio 2023

Indice

1 Introduzione	7
1.1 Sistemi informativi, informazioni e dati	7
1.2 Basi di dati e sistemi di gestione di basi di dati	8
1.3 Linguaggi per basi di dati	9
1.4 Modelli dei dati	10
1.5 Astrazione (architettura) dei DBMS	11
1.6 Indipendenza dei dati	12
2 Metodologie e modelli per il progetto	13
2.1 Ciclo di vita dei sistemi informativi	13
2.2 Metodologie di progettazione e basi di dati	15
2.3 Il modello Entità-Relazione (E-R)	16
2.4 I costrutti principali del modello	17
2.4.1 Entità	17
2.4.2 Relazioni (o associazioni)	18
2.4.3 Attributi	19
2.5 Altri costrutti del modello	20
2.5.1 Cardinalità delle relazioni	20
2.5.2 Cardinalità degli attributi	21
2.5.3 Identificatori	22
2.5.4 Generalizzazioni	24
3 Progettazione concettuale	26
3.1 Strategie di progetto	26
3.1.1 Strategia top-down	26
3.1.2 Strategia bottom-up	27
3.1.3 Strategia inside-out	28
3.2 Qualità di uno schema concettuale	29
3.2.1 Correttezza	29
3.2.2 Completezza	29
3.2.3 Leggibilità	29
3.2.4 Minimalità	30
4 Progettazione logica	31
4.1 Fasi della progettazione logica	31
4.2 Traduzione verso il modello logico	32
4.2.1 Entità e attributo opzionale	32
4.2.2 Relazione uno a molti	33
4.2.3 Relazione uno a uno	34
4.2.4 Relazione molti a molti	35
4.2.5 Relazione uno a molti (identificatore esterno)	36
4.2.6 Relazione uno a molti (attributo sulla relazione)	37
4.2.7 Relazione uno a uno (una cardinalità minima a zero)	38
4.2.8 Relazione uno a uno (entrambe cardinalità minima a zero)	39
4.2.9 Relazione molti a molti (attributo sulla relazione)	40
4.2.10 Relazione molti a molti (identificatori con più attributi)	41
4.2.11 Relazione molti a molti (relazione ternaria)	42

4.2.12 Relazione molti a molti (relazione ternaria e cardinalità uno a uno)	43
5 Algebra relazionale	44
5.1 Insiemistici	45
5.1.1 Unione	45
5.1.2 Intersezione	46
5.1.3 Differenza	46
5.2 Specifici	47
5.2.1 Ridenominazione	47
5.2.2 Selezione	48
5.2.3 Proiezione	50
5.3 Join	52
5.3.1 Join naturale	52
5.3.2 Join completi e incompleti	54
5.3.3 Theta-join ed equi-join	57
5.4 Algebra con valori nulli	60
5.5 Ottimizzare ed equivalenza di espressioni algebriche	62
5.5.1 Definizioni	62
5.5.2 Trasformazioni di equivalenza	63
6 Calcolo relazionale	66
6.1 Calcolo su tuple con dichiarazioni di range	67
6.2 Unione, intersezione e differenza	68
6.3 Esempi	69
6.4 Esercizi	71
6.4.1 Aula, insegnamento, docente e lezione	71
7 Tecnologie per le basi di dati	75
7.1 Transazioni	75
7.2 Transazioni in SQL	75
7.3 Proprietà acide delle transazioni	76
7.3.1 Atomicità	76
7.3.2 Consistenza	77
7.3.3 Isolamento	77
7.3.4 Persistenza	77
7.4 Architettura di riferimento di un DBMS	78
7.4.1 Gestore (ottimizzatore) delle interrogazioni	79
7.4.2 Gestore dei metodi d'accesso e dell'esecuzione concorrente	79
7.4.3 Gestore dei metodi d'accesso e dell'affidabilità	80
7.4.4 Gestore dell'affidabilità e del buffer	80
8 Approfondimento gestore dell'affidabilità e del buffer	81
8.1 Memoria secondaria	81
8.2 Gestione del buffer	82
8.3 Primitive per la gestione del buffer	83
8.3.1 Primitiva fix	83
8.3.2 Primitiva setDirty	84
8.3.3 Primitiva unfix	84
8.3.4 Primitiva force	84

8.4	Gestore dell'affidabilità	85
8.4.1	Definizione log	86
8.4.2	Notazione dei record nel log	87
8.4.3	Checkpoint e dump	88
8.4.4	Esecuzione delle transazioni e scrittura dei log	89
8.4.5	Gestione dei guasti	90
8.4.6	Ripresa a caldo	91
8.4.7	Ripresa a freddo	92
8.4.8	Esercizio sulla ripresa a caldo	93
9	Gestore dei metodi d'accesso	95
9.1	Gestione delle tuple nelle pagine	95
9.2	Strutture primarie per l'organizzazione di file	97
9.3	Strutture sequenziali	97
9.3.1	Struttura sequenziale ordinata	97
9.3.2	Indici primari e secondari	98
9.3.3	Esempi di indice primario e secondario	99
10	Strutture ad albero e hash	100
10.1	Alberi B+ (<i>B+-Tree</i>)	100
10.1.1	Definizione	100
10.1.2	Struttura di un nodo foglia	100
10.1.3	Struttura di un nodo intermedio	101
10.1.4	Vincoli di riempimento	101
10.1.5	Esempio di albero B+	102
10.2	Operazione sugli alberi	103
10.2.1	Ricerca con chiave K	103
10.2.2	Inserimento con chiave K	105
10.2.3	Cancellazione con chiave K	106
10.3	Hash	108
10.3.1	Definizione	108
10.3.2	Operazioni	109
10.3.3	Problema della collisione	109
10.4	Confronto alberi B+ e Hashing	111
10.4.1	Operazione di ricerca	111
10.4.2	Operazione di inserimento e cancellazione	111
11	Controllo di concorrenza	112
11.1	Anomalie delle transazioni concorrenti	112
11.1.1	Perdita di aggiornamento	112
11.1.2	Lettura sporca	113
11.1.3	Letture inconsistenti	114
11.1.4	Aggiornamento fantasma	115
11.2	Teoria del controllo di concorrenza	116
11.2.1	View-equivalenza	117
11.2.2	Conflict-equivalenza	118
11.2.3	Locking a due fasi (2PL)	120
11.3	Blocco critico	126
11.3.1	Timeout	126
11.3.2	Prevenzione (<i>deadlock prevention</i>)	127

11.3.3 Rilevamento (<i>deadlock detection</i>)	128
12 Gestore delle interrogazioni: esecuzione e ottimizzazione	129
12.1 Definizione	129
12.2 Profili delle relazioni	131
12.3 Operazioni di scansione	131
12.4 Ordinamenti	132
12.4.1 Passaggi dell'algoritmo Z-way Sort-Merge	132
12.4.2 Esempio di applicazione dell'algoritmo	133
12.5 Accesso diretto	134
12.6 Metodo di join	135
12.6.1 Nested loop	135
12.6.2 Merge scan	136
12.6.3 Hash-based	137
13 PostgreSQL	138
13.1 Introduzione e installazione pgAdmin 4	138
13.2 Se ti è piaciuto pgAdmin, adorerai VSCode	140
13.3 Caratteri (character, character varying, text)	141
13.4 Booleani (boolean)	142
13.5 Tipi numerici esatti (numeric, decimal, integer, smallint, bigint)	142
13.6 Tipi numerici approssimati (float, real, double precision)	143
13.7 Istanti e intervalli temporali (date, time, timestamp, interval)	143
13.8 Oggetti di grandi dimensioni (blob, clob)	144
13.9 Definizione delle tabelle (create table)	144
13.10 Definizione dei domini (create domain)	145
13.11 Valori di default (default)	145
13.12 Vincoli intrarelazionali	146
13.12.1 Not null	146
13.12.2 Unique	146
13.12.3 Primary key	147
13.13 Vincoli interrelazionali (foreign key)	148
13.13.1 Politiche di reazione ad una modifica/eliminazione	149
13.14 Vincolo di integrità generico (check)	151
13.15 Modifica degli schemi	153
13.15.1 Modifica dei domini e schemi (alter)	153
13.15.2 Rimuovere schemi, domini, viste, asserzioni (drop)	154
13.16 Modifica dei dati	155
13.16.1 Inserimento (insert into)	155
13.16.2 Cancellazione (delete, drop)	156
13.16.3 Modifica (update)	157
13.17 Interrogazioni semplici	158
13.17.1 Select	158
13.17.2 From	160
13.17.3 Where e i suoi operatori (like, similar to, between, in, is null)	161
13.18 Ordinamento (order by)	164
13.19 Operatori aggregati (count, sum, max, min, avg)	165
13.20 Join interni ed esterni (inner/right/left/full/cross join)	167
13.21 Raggruppamento (group by)	170
13.21.1 Predicati sui gruppi (having)	173

13.22	Interrogazioni nidificate (any, all, in, not in, some)	174
13.22.1	Operatori any/some e all	174
13.22.2	Operatori in e not in	175
13.22.3	Interrogazioni nidificate complesse (exists), passaggio di binding e scope	176
13.22.4	Interrogazioni nidificate nelle clausole select e from	179
13.23	Interrogazioni di tipo insiemistico (union, intersect, except)	180
13.24	Viste (create view)	182
13.24.1	Esempio ampio con le viste	184

1 Introduzione

1.1 Sistemi informativi, informazioni e dati

Ogni organizzazione è dotata di un *sistema informativo*, che organizza e gestisce le informazioni necessarie per perseguire gli scopi dell'organizzazione stessa. Per indicare la **porzione automatizzata del sistema informativo** viene di solito utilizzato il termine *sistema informatico*.

Nei sistemi informatici le informazioni vengono rappresentate per mezzo di *dati*, che hanno bisogno di essere interpretati per fornire informazioni. Esiste una differenza sottile tra dato e informazioni. Solitamente i primi, se presi da soli, non hanno significato, ma, una volta interpretati e correlati opportunamente, essi forniscono informazioni, che consentono di arricchire la conoscenza:

Informazione: notizia, dato o elemento che consente di avere conoscenza più o meno esatta di fatti, situazioni, modi di essere;

Dato: ciò che è immediatamente presente alla conoscenza, prima di ogni elaborazione. In informatica, sono elementi di informazione costituiti da simboli che devono essere elaborati.

[ESAME] Definizione base di dati: Una *base di dati* è una collezione di dati, utilizzati per rappresentare con tecnologia informatica le informazioni di interesse per un sistema informativo.

1.2 Basi di dati e sistemi di gestione di basi di dati

Inizialmente, venne adottato un “approccio convenzionale” alla gestione dei dati. Esso **sfruttava** la presenza di archivi o **file per memorizzare e per ricercare dati**. Tuttavia, i metodi di accesso e condivisione erano semplici e banali. Infatti, erano presenti numerosi **problemi**:

- ✗ **Accesso sequenziale:** la scarsa efficienza nell’accesso ai dati su file rendeva lento l’accesso a tali informazioni;
- ✗ **Ridondanza:** i dati di interesse per più programmi sono replicati tante volte quanti sono i programmi che li utilizzano, con evidente ridondanza e possibilità di incoerenza;
- ✗ **Inconsistenza:** una diretta conseguenza della ridondanza. Con la presenza di più copie di un determinato dato, l’eventuale cambiamento di uno solo potrebbe portare a questo effetto;
- ✗ **Progettazione duplicata:** per ogni programma viene replicata la progettazione.

La **soluzione** è arrivata negli anni ’80 con l’avvento delle **basi di dati**. Quest’ultime gestiscono in modo integrato e flessibile le informazioni di interesse per diversi soggetti.

[ESAME] Definizione DBMS: Un *sistema di gestione di basi di dati* (in inglese *Data Base Management System*, **DBMS**) è un sistema software in grado di gestire collezioni di dati che siano:

- ✓ **Grandi;**
- ✓ **Condivise;**
- ✓ **Persistenti.**

assicurando allo stesso tempo:

- ★ **Affidabilità;**
- ★ **Privatezza;**
- ★ **Accesso efficiente.**

Il **vantaggio** di utilizzare un DBMS è stato evidenziato nella definizione. Quindi:

- ✓ **Maggiore astrazione** poiché le sue funzioni estendono il *file system*, fornendo la possibilità di accesso condiviso agli stessi dati da parte di più utenti e applicazioni;
- ✓ **Maggiore efficacia** poiché le operazioni di accesso ai dati si basano su un linguaggio di interrogazione.

1.3 Linguaggi per basi di dati

Su un DBMS è possibile specificare operazioni di vario tipo, ma principalmente si distinguono in due categorie:

- **Linguaggi di definizione dei dati** (*Data Definition Language*, abbreviato con **DDL**) utilizzati per definire gli schemi logici, esterni e fisici e le autorizzazioni per l'accesso;
- **Linguaggi di manipolazione dei dati** (*Data Manipulation Language*, abbreviato con **DML**) utilizzati per l'interrogazione e l'aggiornamento delle istanze di basi di dati:
 - *Linguaggio di interrogazione*, estrae informazioni da una base di dati (SQL, algebra relazionale);
 - *Linguaggio di manipolazione*, popola la base di dati, modifica il suo contenuto con aggiunte, cancellazioni e variazioni sui dati (SQL).

1.4 Modelli dei dati

Definizione modello dei dati: Un *modello dei dati* è un insieme di concetti utilizzati per organizzare i dati di interesse e descriverne la struttura in modo che essa risulti comprensibile ad un elaboratore.

Ogni modello dei dati fornisce **meccanismi di strutturazione**, analoghi ai **costruttori** di tipo dei linguaggi di programmazione (es: Java), che permettono di definire nuovi tipi sulla base di tipi predefiniti (elementari) e costruttori di tipo. Quindi, i *costruttori* consentono di:

- ☞ **Definire** le strutture dati che conterranno le informazioni della base di dati;
- ☞ **Specificare** le proprietà che dovranno soddisfare le istanze di informazione che saranno contenute nelle strutture dati.

Definizione schemi e istanze: È molto importante distinguere gli **schemi** e le **istanze** dal concetto di modello dei dati:

- **Schema:** parte invariante nel tempo, è costituita dalle caratteristiche dei dati. In altre parole, è la descrizione della struttura e delle proprietà di una specifica base di dati fatta utilizzando i costruttori del modello dei dati;
- **Istanza o stato:** parte variabile nel tempo, è costituita dai valori effettivi. Quest'ultimi, in un certo istante, popolano le strutture dati della base di dati.

Modello dei dati	Schema	Istanza						
Basi di dati Tabella (o relazione)	P(cognome: VARCHAR(40), nome: VARCHAR(30))	<table border="1"> <thead> <tr> <th>cognome</th><th>nome</th></tr> </thead> <tbody> <tr> <td>Rossi</td><td>Mario</td></tr> <tr> <td>Bianchi</td><td>Lisa</td></tr> </tbody> </table>	cognome	nome	Rossi	Mario	Bianchi	Lisa
cognome	nome							
Rossi	Mario							
Bianchi	Lisa							
Linguaggi di progr. Array	<pre>Class Persona { String cognome; String nome; } Class X { ... Persona[] p; p = new Persona[100]; }</pre>	<p>Diagram illustrating the relationship between the array 'p' and the 'Persona' objects it contains. The array 'p' is shown as a yellow rectangle with three horizontal bars inside, representing an array of Persona objects. An arrow points from the variable 'p' to this array. Another arrow points from the array to a specific 'Persona' object, which is shown as a red rectangle with two fields: 'Nome: Mario' and 'Cognome: Rossi'.</p>						

Figura 1: Esempio di modello di dati, schema e istanza.

1.5 Astrazione (architettura) dei DBMS

Esiste un'architettura standardizzata per i DMBS, la quale si caratterizza su tre livelli: **esterno**, **logico** e **interno**:

- ★ **Schema logico.** È la rappresentazione della **struttura** e delle **proprietà** della **base di dati** definita attraverso i costrutti del modello dei dati del DBMS. In altre parole, descrive l'intera base di dati per mezzo del modello logico adottato dal DBMS (quindi relazione o ad oggetti).
- ★ **Schema interno.** È la rappresentazione della base di dati per mezzo delle **strutture fisiche di memorizzazione** (e.g. file sequenziale, file hash, ecc.).
- ★ **Schema esterno.** Descrive una **porzione dello schema logico** di interesse per uno **specifico** utente o applicazione. Possono esistere più schemi esterni che consentono di avere punti di vista differenti senza cambiare la logica di base.

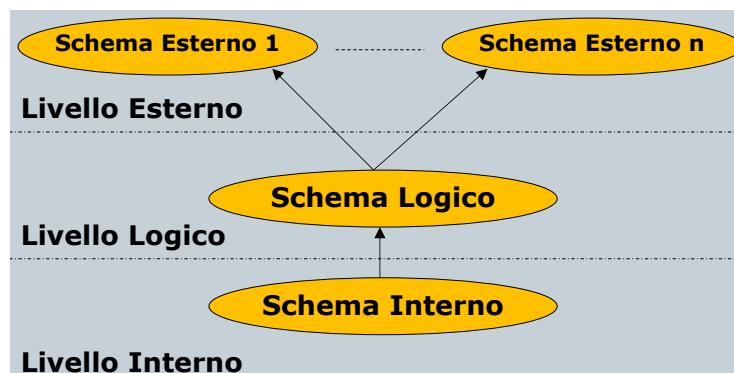


Figura 2: Architettura generale di un DBMS.

1.6 Indipendenza dei dati

L'architettura a livelli definita nel paragrafo 1.5 garantisce l'**indipendenza dei dati**, la **proprietà più importante** dei DBMS. L'**obbiettivo** è quello di poter fornire all'utente una basi di dati in grado di interagire con un elevato livello di astrazione. Esistono due tipi di indipendenza:

- ☛ **Indipendenza fisica.** Lo schema logico della basi di dati è completamente indipendente dallo schema interno. Quindi, l'interazione con il DBMS può essere effettuato in modo indipendente dalla struttura fisica dei dati.
Vantaggio: le modifiche non influiscono sullo schema logico, cioè sulle applicazioni che lo utilizzano.
- ☛ **Indipendenza logica.** Gli schemi esterni (definizione nel paragrafo: 1.5) della base di dati sono indipendenti dallo schema logico. Quindi, è possibile interagire con il livello esterno in modo indipendente dal livello logico.
Vantaggio:
 - I **Aggiunta/Modifica** di uno schema esterno in base alle esigenze di un nuovo utente, senza modificare lo schema logico;
 - II **Modifica** di uno schema logico mantenendo inalterate le strutture esterne.

2 Metodologie e modelli per il progetto

2.1 Ciclo di vita dei sistemi informativi

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

- ☛ **Studio di fattibilità.** Definisce i costi delle varie alternative possibili e stabilisce le priorità di realizzazione delle varie componenti del sistema.
- ☛ **Raccolta e analisi dei requisiti.** Individua le proprietà e le funzionalità che il sistema informativo deve avere producendo una descrizione completa, ma generalmente informale.
- ☛ **Progettazione.** Si divide in due fasi:
 - **Progettazione dei dati.** Individua la struttura e l'organizzazione che i dati devono avere.
 - **Progettazione delle applicazioni.** Definizione delle caratteristiche dei programmi applicativi.
- ☛ **Implementazione (su un DBMS).** È la realizzazione del sistema informativo secondo la struttura e le caratteristiche fornite durante la fase di progettazione. In questa fase viene costruita e popola la base di dati.
- ☛ **Validazione e collaudo.** Verifica il corretto funzionamento e la qualità del sistema informativo.
- ☛ **Funzionamento.** Il sistema informativo diventa operativo ed esegue i compiti per i quali è stato progettato.

Spesso il processo **non** è strettamente sequenziale. Infatti, come si vede dalla seguente figura, durante l'esecuzione di una delle attività sopraelencate, è necessario rivedere decisioni prese nell'attività precedente.

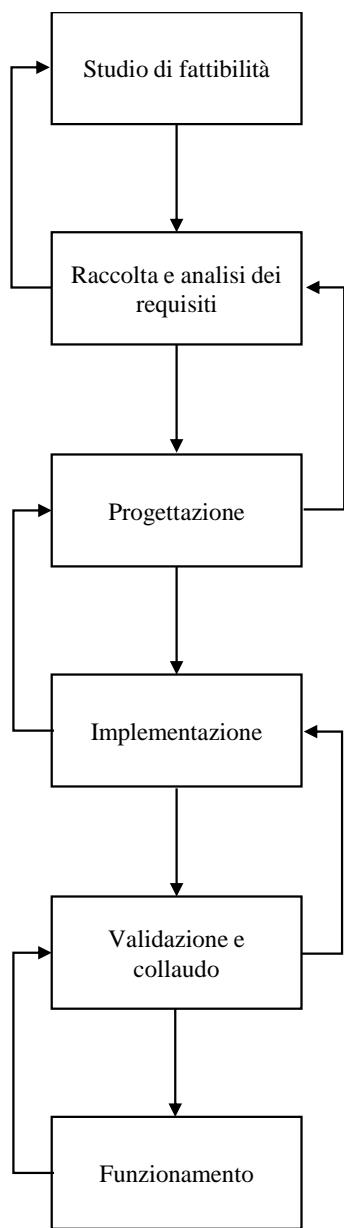


Figura 3: Ciclo di vita di un sistema informativo.

2.2 Metodologie di progettazione e basi di dati

Una **metodologia di progettazione** consiste in:

- ✓ **Decomposizione** dell'intera attività di progetto in passi successivi indipendenti tra loro.
- ✓ **Strategie** da seguire nei vari passi e **criteri** nel caso di alternative.
- ✓ **Modelli di riferimento** per descrivere i dati in ingresso e uscita delle varie fasi.

Le **proprietà** che una metodologia deve garantire sono:

- ★ **Generalità** rispetto alle applicazioni e ai sistemi in gioco;
- ★ **Qualità del prodotto** in termini di correttezza, completezza ed efficienza rispetto alle risorse impiegate;
- ★ **Facilità d'uso** delle strategie e dei modelli di riferimento.

Negli anni si è *consolidata una metodologia* di progetto che ha dato prova di soddisfare pienamente le proprietà descritte. Si basa sull'idea di separare le decisioni relative a "cosa" rappresentare in una base di dati (prima fase), da quelle relative a "come" farlo (seconda e terza fase):

● Progettazione concettuale.

Obiettivo: rappresentare le specifiche informali della realtà di interesse in termini di una descrizione formale e completa. La **rappresentazione deve essere indipendente** dai criteri di rappresentazione utilizzati nei sistemi di gestione di basi di dati.

Prodotto di questa fase: schema concettuale. È un documento formale che rappresenta il contenuto della base di dati in modo indipendente dall'implementazione (DBMS).

Applicazione: cercare di rappresentare il **contenuto informativo** della base di dati, senza preoccuparsi né della 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.

Obiettivo: traduzione dello schema concettuale prodotto nella fase precedente, in termini del modello di rappresentazione dei dati adottato dal sistema di gestione di base di dati a disposizione.

Prodotto di questa fase: schema logico.

Applicazione: durante la traduzione, le scelte progettuali si devono basare anche su criteri di ottimizzazione delle operazioni da effettuare sui dati.

● Progettazione fisica.

Obiettivo: lo schema logico viene completato con la specifica dei parametri fisici di memorizzazione dei dati.

Prodotto di questa fase: schema fisico.

2.3 Il modello Entità-Relazione (E-R)

Il **modello Entità-Relazione** è un modello **concettuale** di dati, quindi utilizzato nella **progettazione concettuale**, e fornisce una serie di strutture, chiamati **costrutti**, atte a descrivere la realtà di interesse in una maniera facile da comprendere e che prescinde dai criteri di organizzazione dei dati nei calcolatori.

I costrutti vengono utilizzati per **definire schemi** che descrivono l'**organizzazione e la struttura delle occorrenze** (o **istanze**) dei **dati**, ovvero, dei valori assunti dai dati al variare del tempo.

Si possono riassumere le caratteristiche del modello Entità-Relazione:

- ☛ **Modello concettuale.** Utilizzato durante la progettazione concettuale (definizione al paragrafo 2.2) di una base di dati.
- ☛ **Strumenti formali.** Vengono messi a disposizione diversi strumenti per definire la **struttura** e le **proprietà** di una base di dati (*esempio i costrutti*).
- ☛ **Indipendente dalla tecnologia.** Essendo un modello astratto, l'obiettivo è quello di definire la struttura e le proprietà della base di dati (non di implementarla!).
- ☛ **Formale.** È facile da utilizzare nonostante non ammetta ambiguità.
- ☛ **Grafico.** La sintassi è prettamente grafica e questo aumenta anche la leggibilità.

2.4 I costrutti principali del modello

Si analizzano i principali costrutti di questo modello: entità (pagina 17), relazioni (pagina 18) e attributi (pagina 19).

2.4.1 Entità

Definizione. Rappresentano **classi di oggetti** (per esempio, fatti, cose, persone) che hanno proprietà comuni ed esistenza “autonoma” ai fini dell’applicazione di interesse. Per esempio, “città, dipartimento, impiegato, acquisto e vendita” sono entità di un’applicazione aziendale. Inoltre, **ogni entità ha un nome identificativo**, il quale deve essere **univoco**. In sintesi:

- Hanno proprietà comuni;
- Hanno esistenza autonoma;
- Hanno identificazione univoca.

Sintassi grafica.



Figura 4: Sintassi grafica dell’entità.

Istanza (o occorrenza). Un’istanza (o occorrenza) di un’entità è un **oggetto della classe che l’entità rappresenta**. Le città di Roma, Milano e Palermo sono esempi di occorrenze dell’entità “Città”.

Attenzione! L’istanza di un’entità *non è un valore* che identifica un oggetto (per esempio, il cognome dell’impiegato o il suo codice fiscale), *ma è l’oggetto stesso* (l’impiegato in “carne e ossa”). Quindi, un’istanza ha un’esistenza indipendente dalle proprietà a esso associate.

Un’istanza dell’entità E è un oggetto appartenente alla classe rappresentata da E . Si indica con $I(E)$ l’insieme delle istanze di E che esistono nella base di dati in un certo istante.

2.4.2 Relazioni (o associazioni)

Definizione. Rappresentano **legami logici**, significativi per l'applicazione di interesse, **tra due o più entità**. Per esempio, “Residenza” è una relazione che sussiste tra le entità “Città” e “Impiegato”. Nello schema E-R, **ogni relazione ha un nome identificativo univoco**.

Le relazioni possono essere di **tipo**:

- **Ricorsive**, ovvero **relazioni tra un'entità e se stessa**. Per esempio, la relazione “Collega” sull'entità “Impiegato” connette coppie di impiegati che lavorano insieme.
- **n-arie**, ovvero **relazioni che coinvolgono più di due entità**. Per esempio, la relazione “Fornitura” tra le tre entità “Fornitore, Prodotto e Dipartimento” descrive il fatto che un fornitore rifornisce un dipartimento di un certo prodotto.

Nota fondamentale: per eseguire una relazione, le entità devono essere tutte piene o con almeno un dato all'interno.

Sintassi grafica. Una relazione R si rappresenta nello schema con un **rombo** a cui si collegano attraverso linee spezzate le entità coinvolte nella relazione. Il nome della relazione viene scritto a fianco del rombo.

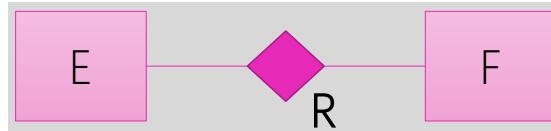


Figura 5: Sintassi grafica della relazione.

Istanza (o occorrenza). Un'istanza di relazione è un'**ennupla costituita da istanze di entità, una per ciascuna delle entità coinvolte**.

Data una relazione R tra n entità E_1, \dots, E_n , un'istanza della relazione R è una ennupla di istanze di entità:

$$(e_1, \dots, e_n) \text{ dove } e_i \in I(E_i), 1 \leq i \leq n$$

Infine, esiste una relazione importante. Data una relazione R tra n entità, vale sempre la seguente proprietà sull'insieme delle istanze di $R(I(R))$:

$$I(R) \subseteq I(E_1) \times \dots \times I(E_n)$$

2.4.3 Attributi

Definizione. Descrivono le **proprietà elementari** di entità o relazioni che sono di interesse ai fini dell'applicazione. Per esempio, "Cognome, Stipendio ed Età" sono possibili attributi dell'entità "Impiegato".

Un attributo associa a ciascun istanza di entità (o relazione) **uno e un solo** valore appartenente a un insieme, chiamato **dominio**, che contiene i valori ammissibili per l'attributo. Per esempio, l'attributo "Cognome" dell'entità "Impiegato" può avere come dominio l'insieme delle stringhe di 20 caratteri.

L'attributo può essere visto come una **funzione che ha come dominio le istanze dell'entità** (o relazione) e come **codominio l'insieme dei valori ammissibili**:

$$f_a : I(E) \rightarrow D$$

- a è un attributo dell'entità E ;
- $I(E)$ è l'insieme delle istanze di E ;
- D è l'insieme dei valori ammissibili.

Sintassi grafica.

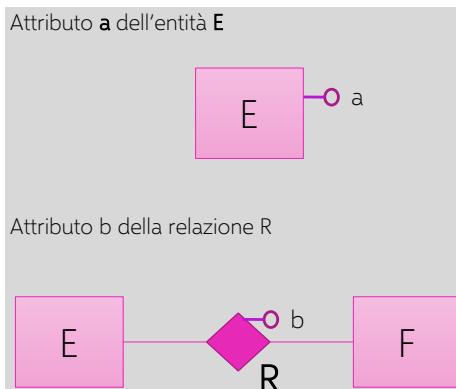


Figura 6: Sintassi grafica dell'attributo.

Istanza (o occorrenza). Dato un attributo a di un'entità E (o relazione R), un'istanza di a è il valore v che esso assume su un'istanza di E (o istanza di R).

Quindi, data un'istanza e dell'entità E (o relazione R), l'istanza di un suo attributo a si ottiene dalla funzione f_a applicata a e :

$$\text{valore di } a \text{ su } e = f_a(e)$$

Attributi composti.

Questo tipo di attributo viene introdotto solo a fini didattici, ma l'obiettivo è quello di usare unicamente gli attributi normali. Talvolta potrebbe tornare comodo raggruppare **attributi di una medesima entità o relazione che presentano affinità nel loro significato o uso**: tale insieme prende il nome di **attributo composto**. Per esempio, gli attributi "Via, Numero civico e CAP" dell'entità "Persona" per formare l'attributo composto "Indirizzo".

2.5 Altri costrutti del modello

I rimanenti costrutti del modello E-R sono le cardinalità delle relazioni e degli attributi e gli identificatori.

2.5.1 Cardinalità delle relazioni

Definizione. Le **cardinalità** vengono specificate per ciascuna entità collegata ad una relazione e descrivono il **numero minimo e massimo di occorrenze** di relazione a cui una occorrenza dell'entità può partecipare.

Più formalmente, data una relazione R i vincoli di cardinalità vengono specificati per ogni entità E_i coinvolta nella relazione R e specificano: il numero massimo e il numero minimo di istanze di R a cui un'istanza di E_i deve/può partecipare.

In parole povere, dicono quante volte, in una relazione tra entità, un'istanza di una di queste entità può essere legata a istanze delle altre entità coinvolte. **Per esempio**, in una relazione “Assegnamento” tra le entità “Impiegato” e “Incarnico” si specifica per la prima entità una cardinalità minima pari a uno e una cardinalità massima pari a cinque. Quindi, un impiegato può partecipare a un minimo di una occorrenza e a un massimo di cinque occorrenze della relazione “Assegnamento”.

N.B. Specificando **zero come cardinalità minima**, si impone che un'occocrenza può apparire oppure no.

È possibile **assegnare un qualunque valore intero non negativo a una cardinalità di una relazione con l'unico vincolo che la cardinalità minima deve essere minore o uguale della cardinalità massima**.

Tuttavia, nella maggior parte dei casi, è sufficiente utilizzare solamente tre simboli: 0, 1 e N (molti).

★ Cardinalità minima.

- **Zero.** La partecipazione dell'entità relativa è *opzionale*;
- **Uno.** La partecipazione dell'entità relativa è *obbligatoria*.

★ Cardinalità massima.

- **Uno.** La partecipazione dell'entità relativa è come una funzione che associa a una occorrenza dell'entità una sola occorrenza (o nessuna) dell'altra entità che partecipa alla relazione;
- **N (molti).** Esiste un'associazione con un numero arbitrario di occorrenze dell'altra entità.

Sintassi grafica.

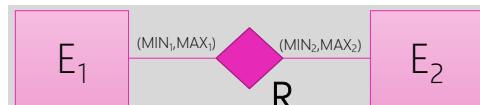


Figura 7: Sintassi grafica della cardinalità.

2.5.2 Cardinalità degli attributi

Definizione. Le **cardinalità degli attributi** è specificata per gli attributi di entità o relazione e hanno l'obiettivo di **descrivere il numero minimo e massimo di valori dell'attributo associati a ogni occorrenza di entità o relazione**.

Solitamente, il valore di cardinalità pari $(1, 1)$, ma si possono avere vari casi:

- $(1, 1)$ —> L'attributo rappresenta una funzione che associa ad ogni occorrenza di entità un solo valore dell'attributo. Solitamente viene omesso e, come si vede nell'immagine 8, la “Persona” ha uno e un solo “Cognome”;
- $(0, 1)$ —> L'attributo con cardinalità minima pari a zero vuol dire che è **opzionale** e la cardinalità massima pari a uno indica che nel **caso in cui esista**, questo valore è **unico**. Nell'esempio in figura 8, la persona può avere solo un numero di patente, ma potrebbe anche non avercela;
- $(1, N)$ —> L'attributo deve esistere, ma contiene più valori, quindi si dice che è **multivaleore**;
- $(0, N)$ —> L'attributo è opzionale, ma se esiste può essere multivaleore.

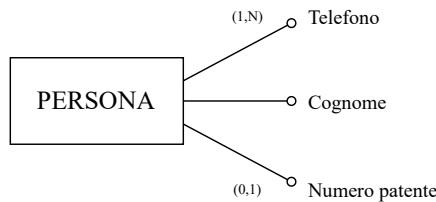


Figura 8: Esempio di cardinalità degli attributi.

2.5.3 Identificatori

Definizione. Vengono specificati per ciascuna entità di uno schema e descrivono i concetti (attributi e/o entità) dello schema che permettono di identificare in maniera univoca le occorrenze delle entità.

È assolutamente vietato inserire uno o più identificatori all'interno di una relazione. Quindi, quest'ultima non può avere identificatori interni!

Per esempio, un identificato interno per l'entità “Automobile” con attributi “Modello, Targa e Colore” è l'attributo “Targa”, in quanto non possono esistere due automobili con la stessa targa e quindi due occorrenze dell'entità “Automobile” con gli stessi valori sull'attributo “Targa”.

Un'entità E può essere identificata da altre entità solo se tali entità sono coinvolte in una relazione a cui E partecipa con cardinalità (1, 1). Nei casi in cui l'identificazione di un'entità è ottenuta utilizzando altre entità si parla di **identificatore esterno**.

Per comprendere meglio si espone un **esempio**. Per identificare univocamente uno studente serve, oltre al numero di matricola, anche la relativa università. Quindi, un identificatore corretto per l'entità “Studente” in questo schema è costituito dall'attributo “Matricola” e dall'entità “Università”.

Quindi, in generale:

- Un identificatore può **coinvolgere uno o più attributi**, ognuno dei quali deve avere cardinalità (1, 1);
- Un identificatore esterno può **coinvolgere una o più entità**, ognuna delle quali deve essere membro di una relazione alla quale l'entità da identificare partecipa con cardinalità (1, 1);
- Un identificatore esterno può **coinvolgere un'entità che è a sua volta identificata esternamente**, purché non vengano generati, in questa maniera, cicli di identificazione esterna;
- Ogni entità deve avere almeno un identificatore (interno o esterno), ma ne può avere in generale più di uno.

Sintassi grafica.

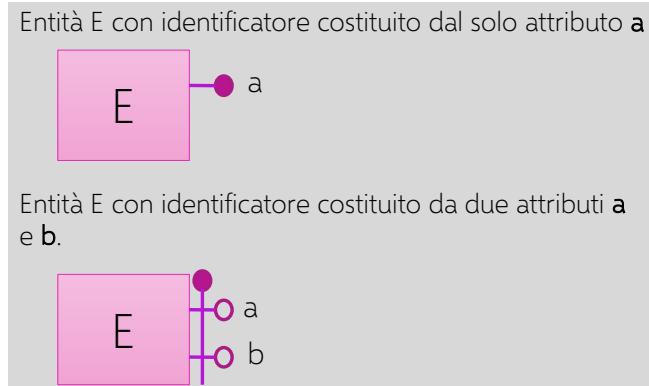


Figura 9: Sintassi grafica dell'identificatore.

2.5.4 Generalizzazioni

Definizione. Sono i legami logici tra un'entità E , chiamata **entità genitore**, e una o più entità E_1, \dots, E_n , dette **entità figlie**, di cui E è più generale, nel senso che le comprende come caso particolare. Quindi, si dice che E è **generalizzazione** di E_1, \dots, E_n e che le entità E_1, \dots, E_n sono **specializzazioni** dell'entità E .

Per esempio, l'entità “Persona” è una generalizzazione delle entità “Uomo e Donna”. Invece, “Professionista” è una generalizzazione delle entità “Ingegnere, Medico e Avvocato”.

Proprietà.

- **Ogni occorrenza di un'entità figlia è anche un'occorrenza dell'entità genitore.** Per esempio, una occorrenza dell'entità “Avvocato” è anche una occorrenza dell'entità “Professionista”.
- **Ogni proprietà dell'entità genitore** (come attributi, identificatori, relazioni e altre generalizzazioni) **è anche una proprietà delle entità figlie.** Per esempio, se l'entità “Persona” ha attributi “Cognome ed Età”, anche le entità “Uomo” e “Donna” possiedono questi attributi.

Classificazioni. Le generalizzazioni possono essere classificate:

- **Totale.** Ogni occorrenza dell'entità genitore è una occorrenza di almeno una dell'entità figlie. Se non è così, la generalizzazione è **parziale**;
- **Esclusiva.** Ogni occorrenza dell'entità genitore è al massimo un'occorrenza di una delle entità figlie. Se non è così, la generalizzazione è **sovraposta**.

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

Sintassi grafica. Non è difficile da comprendere, ma si presti attenzione a (x, y) . Esse indicano il **tipo di generalizzazione**:

- $[t, e] \rightarrow$ totale ed esclusiva;
- $[t, s] \rightarrow$ totale e sovrapposta;
- $[p, e] \rightarrow$ parziale ed esclusiva;
- $[p, s] \rightarrow$ parziale e sovrapposta.

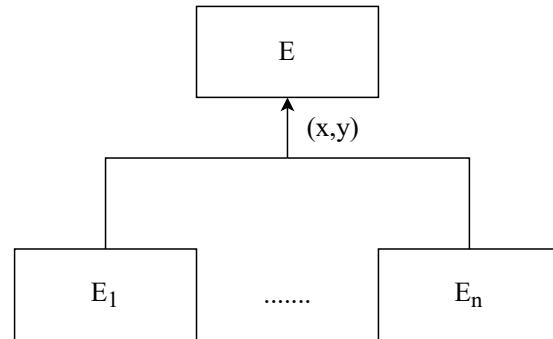


Figura 10: Sintassi grafica della generalizzazione.

3 Progettazione concettuale

3.1 Strategie di progetto

Lo sviluppo di uno schema concettuale a partire dalle sue specifiche può essere considerato un processo di ingegnerizzazione.

3.1.1 Strategia top-down

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

In sintesi:

1. **Fase 1**, si considerano le *specifiche globalmente* e si produce uno schema iniziale completo ma con *pochi concetti molto astratti*;
2. **Fase 2**, si esegue un *raffinamento* dei concetti astratti fino ad arrivare allo schema concettuale *completo* in ogni *dettaglio*.

Vantaggio: il progettista può **descrivere inizialmente tutte le specifiche dei dati trascurandone i dettagli**, per **poi entrare nel merito** di un concetto alla volta.

Svantaggio: si deve **possedere una visione globale e astratta** di *tutte* le componenti del sistema, ma solitamente è difficile.

3.1.2 Strategia bottom-up

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

La **differenza rispetto alla strategia top-down** è che i vari concetti presenti nello schema finale vengono via via introdotti durante le varie fasi.

In sintesi:

1. **Fase 1**, si *decompongono* le specifiche iniziali in *parti elementari*, ovvero in frasi che descrivono lo stesso concetto;
2. **Fase 2**, si *generano* gli schemi per tutte le parti elementari individuate;
3. **Fase 3**, si *fondono* gli schemi (introducendo altri costrutti del modello E-R) in modo da *integrare* tutti gli schemi componenti e generare lo *schema finale*.

Vantaggio: questa strategia si adatta a una **decomposizione del problema in componenti più semplici**, facilmente individuabili, il cui progetto può essere affrontato anche da progettisti diversi.

Svantaggio: vengono richieste delle operazioni di **integrazione di schemi concettuali diversi** che, nel caso di schemi complessi, presentano **quasi sempre grosse difficoltà**.

3.1.3 Strategia inside-out

Si individuano inizialmente solo alcuni concetti importanti e poi si procede, a partire da questi, a “macchia d’olio”. Si rappresentano cioè prima i concetti in relazione con i concetti iniziali, per poi muoversi verso quelli più lontani attraverso una “navigazione” tra le specifiche.

In sintesi:

1. **Fase 1**, si *individua* nelle specifiche alcuni *concetti importanti*, chiamati concetti guida;
2. **Fase 2**, si *generano* gli schemi per i concetti guida;
3. **Fase 3**, si *fondono* gli schemi precedenti e si genera lo *schemma finale*.

Vantaggio: non sono richiesti passi di integrazione.

Svantaggio: è necessario, di volta in volta, esaminare tutte le specifiche per individuare concetti non ancora rappresentati e descrivere i nuovi concetti nel dettaglio.

3.2 Qualità di uno schema concettuale

L'analisi della qualità dello schema concettuale prodotto può essere suddivisa in diverse fasi:

- Correttezza
- Completezza
- Leggibilità
- Minimalità

3.2.1 Correttezza

Definizione. Uno schema concettuale è **corretto** quando **utilizza propriamente i costrutti** messi a disposizione dal modello concettuale di riferimento.

Gli errori che si possono commettere nello schema concettuale sono principalmente due:

- **Sintattici.** Vengono utilizzati costrutti non ammessi. Per esempio, una generalizzazione tra relazioni invece che tra entità.
- **Semantici.** Vengono usati costrutti senza rispettare la loro definizione. Per esempio, l'uso di una relazione per descrivere il fatto che una entità è specializzazione di un'altra.

3.2.2 Completezza

Definizione. Uno schema concettuale è **completo** quando **rappresenta tutti i dati** di interesse e quando **tutte le operazioni possono essere eseguite** a partire dai concetti descritti nello schema.

La completezza è **possibile verificare** controllando che tutte le specifiche sui dati siano rappresentate da qualche concetto presente nello schema che stiamo costruendo, e che tutti i concetti coinvolti in un'operazione presente nelle specifiche siano raggiungibili “navigando” attraverso lo schema.

3.2.3 Leggibilità

Definizione. Uno schema concettuale è **leggibile** quando rappresenta i requisiti in maniera naturale e facilmente comprensibile.

Per garantire questa proprietà è necessario rendere lo schema autoesplicativo, per esempio, mediante una scelta opportuna dei nomi da dare ai concetti. La leggibilità dipende anche da criteri puramente estetici.

3.2.4 Minimalità

Definizione. Uno schema concettuale è **minimale** quando tutte le specifiche sui dati sono rappresentate una sola volta nello schema.

Uno schema quindi non è minimale quando esistono delle **ridondanze**, ovvero concetti che possono essere derivati da altri. La minimalità di uno schema si può **verificare** per ispezione, controllando se esistono concetti che possono essere eliminati dallo schema che stiamo costruendo senza inficiare la sua completezza.

4 Progettazione logica

4.1 Fasi della progettazione logica

L'**obiettivo** della progettazione logica è **produrre uno schema logico che descriva in modo corretto ed efficace tutte le informazioni contenute nello schema concettuale.**

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

- **Ristrutturazione dello schema Entità-Relazione:** è una fase indipendente dal modello logico scelto e si basa su **criteri di ottimizzazione** dello schema e di **semplificazione** della fase successiva. In particolare, le fasi sono:
 - Analisi delle ridondanze dovute alla presenza di dati derivabili;
 - Eliminazione delle generalizzazioni;
 - Accorpamento/partizionamento di entità e relazioni;
 - Scelta degli identificatori principali.
- **Traduzione verso il modello logico:** fa riferimento a uno specifico modello logico (nel nostro caso modello relazionale) e può includere una ulteriore ottimizzazione che si basa sulle caratteristiche del modello logico stesso.

4.2 Traduzione verso il modello logico

La seconda fase della progettazione logica corrisponde ad una traduzione tra modelli di dati diversi. Si parte da uno schema E-R ristrutturato e si costruisce uno schema logico equivalente, in grado cioè di rappresentare le medesime informazioni.

Si affronta il problema della traduzione caso per caso, iniziando dal caso più generale che ci suggerisce l'idea generale su cui si basa la metodologia di traduzione.

4.2.1 Entità e attributo opzionale

L'**entità** si rappresenta una relazione (lettera maiuscola) con lo stesso nome avente per attributi (lettere tra parentesi) i medesimi attributi dell'entità e per chiave il suo identificatore.

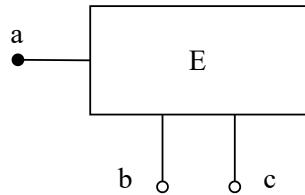


Figura 11: Modello E-R di un'entità.

Relativo modello relazionale:

$$\mathbf{E}(a, b, c)$$

Invece, un possibile **attributo nullo** si rappresenta inserendo un asterisco.

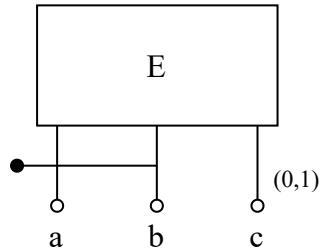


Figura 12: Modello E-R di un possibile attributo nullo.

Relativo modello relazionale:

$$\mathbf{E}(\underline{a}, \underline{b}, c^*)$$

4.2.2 Relazione uno a molti

La **relazione uno a molti** è caratterizzata dal fatto che un'entità è in relazione con un'altra con cardinalità $(1, 1)$ e la corrispondente entità ha cardinalità (x, N) (x può essere qualsiasi valore minimo).

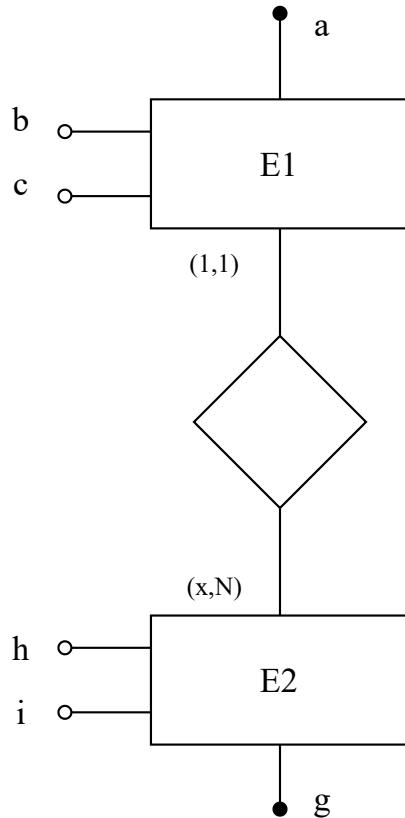
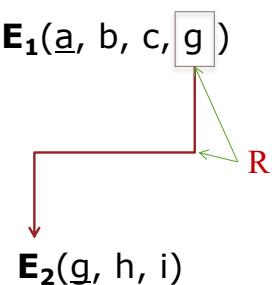


Figura 13: Modello E-R di una relazione uno a molti.

Relativo modello relazionale:



4.2.3 Relazione uno a uno

La relazione uno a uno è caratterizzata dal fatto che entrambe le entità hanno cardinalità (1, 1).

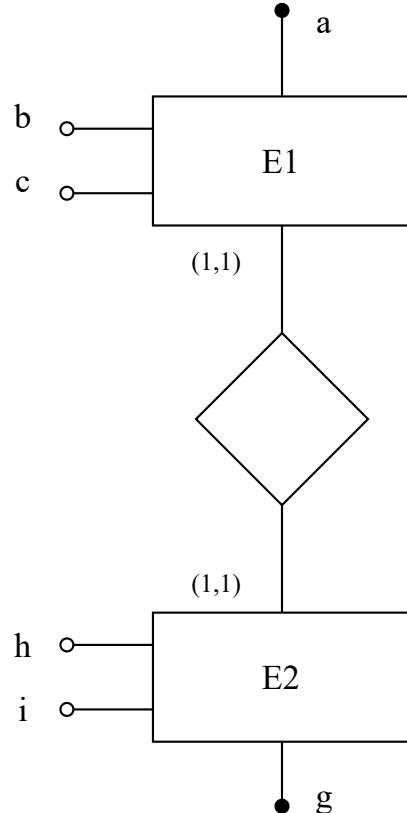
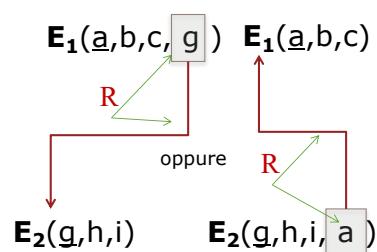


Figura 14: Modello E-R di una relazione uno a uno.

Relativo modello relazionale:



4.2.4 Relazione molti a molti

La **relazione molti a molti** è caratterizzata dal fatto che entrambe le entità hanno cardinalità (x, N) (x può essere qualsiasi valore minimo).

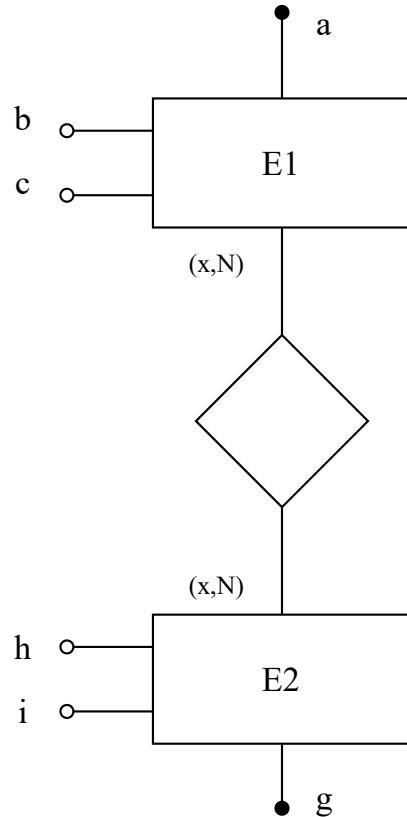
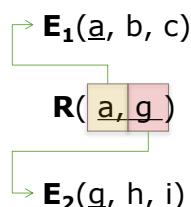


Figura 15: Modello E-R di una relazione molti a molti.

Relativo modello relazionale:



4.2.5 Relazione uno a molti (identificatore esterno)

La relazione uno a molti con identificatore esterno è caratterizzata dal fatto che un'entità ha cardinalità $(1, 1)$ e l'altra entità ha (x, N) (x può essere qualsiasi valore minimo).

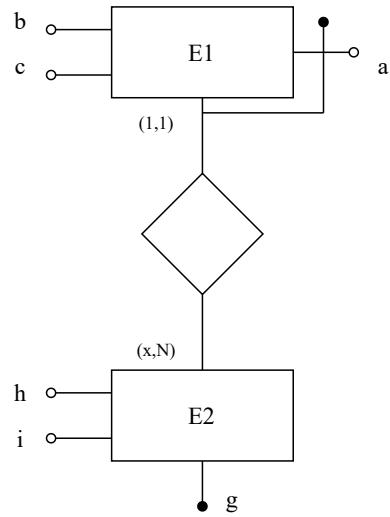
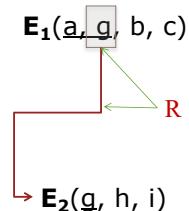


Figura 16: Modello E-R di una relazione molti a molti.

Relativo modello relazionale:



4.2.6 Relazione uno a molti (attributo sulla relazione)

La relazione uno a molti con un attributo sulla relazione è caratterizzata dal fatto che un'entità ha cardinalità $(1, 1)$, l'altra entità ha (x, N) (x può essere qualsiasi valore minimo) e un attributo è presente nella relazione.

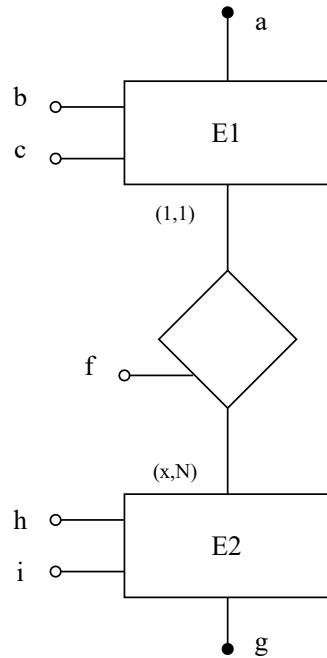
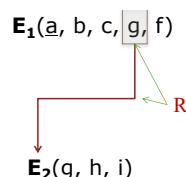


Figura 17: Modello E-R di una relazione uno a molti con un attributo sulla relazione esterno.

Relativo modello relazionale:



4.2.7 Relazione uno a uno (una cardinalità minima a zero)

La relazione uno a uno con una cardinalità minima uguale a zero è caratterizzata dal fatto che un'entità ha cardinalità $(0, 1)$, l'altra entità ha $(1, 1)$.

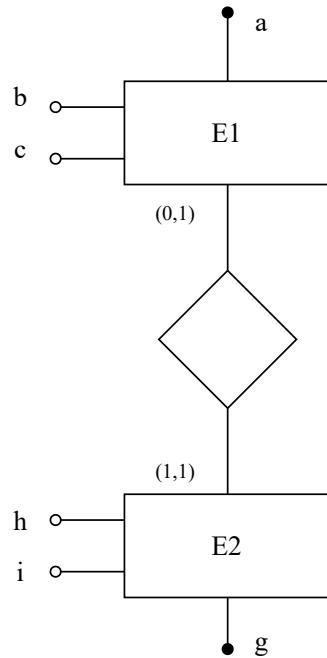
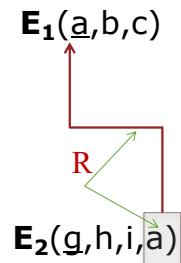


Figura 18: Modello E-R di una relazione uno a uno con una cardinalità minima uguale a zero.

Relativo modello relazionale:



4.2.8 Relazione uno a uno (entrambe cardinalità minima a zero)

La **relazione uno a uno con entrambe le cardinalità minima uguale a zero** è caratterizzata dal fatto che un'entità ha cardinalità $(0, 1)$, l'altra entità ha $(0, 1)$.

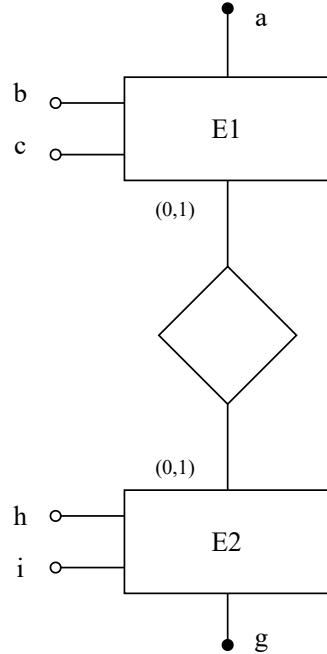
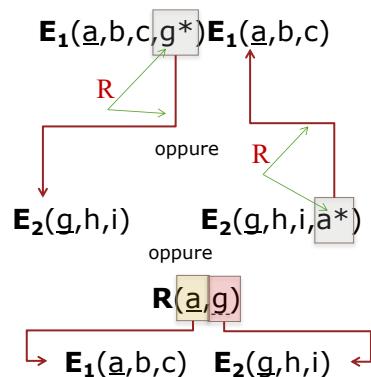


Figura 19: Modello E-R di una relazione uno a uno con entrambe le cardinalità minima uguale a zero.

Relativo modello relazionale:



4.2.9 Relazione molti a molti (attributo sulla relazione)

La **relazione molti a molti con un attributo sulla relazione** è caratterizzata dal fatto che un'entità ha cardinalità (x, N) , l'altra entità ha (x, N) (x può essere qualsiasi valore minimo) e un attributo è presente nella relazione.

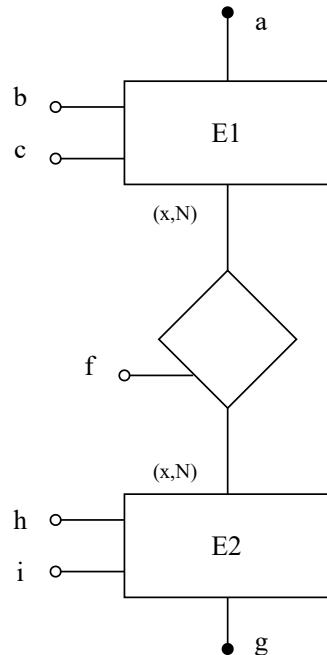
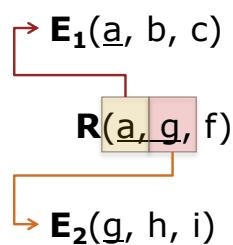


Figura 20: Modello E-R di una relazione molti a molti con un attributo sulla relazione esterno.

Relativo modello relazionale:



4.2.10 Relazione molti a molti (identificatori con più attributi)

La **relazione molti a molti con identificatori con più attributi** è caratterizzata dal fatto che un'entità ha cardinalità (x, N) , l'altra entità ha (x, N) (x può essere qualsiasi valore minimo) e un attributo è presente nella relazione.

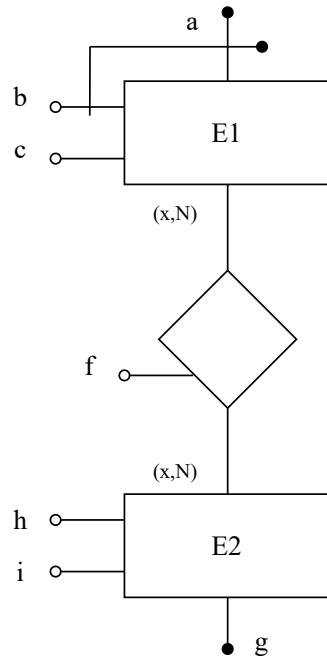
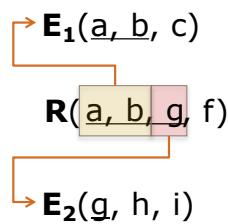


Figura 21: Modello E-R di una relazione molti a molti con identificatori con più attributi.

Relativo modello relazionale:



4.2.11 Relazione molti a molti (relazione ternaria)

La relazione molti a molti con relazione ternaria è caratterizzata dal fatto che un'entità ha cardinalità (x, N) , l'altra entità ha (x, N) , l'altra entità ancora ha cardinalità (x, N) (x può essere qualsiasi valore minimo) e un attributo è presente nella relazione.

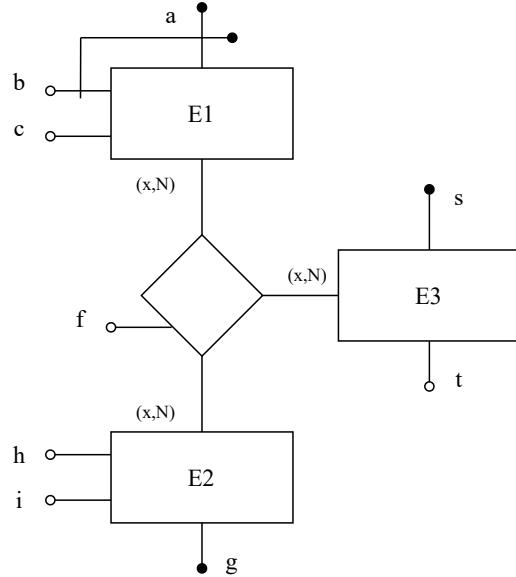
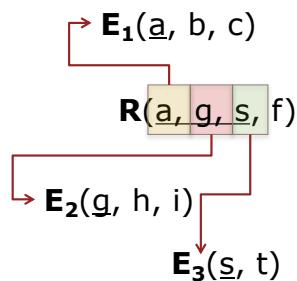


Figura 22: Modello E-R di una relazione molti a molti con relazione ternaria.

Relativo modello relazionale:



4.2.12 Relazione molti a molti (relazione ternaria e cardinalità uno a uno)

La relazione molti a molti con relazione ternaria e cardinalità uno a uno è caratterizzata dal fatto che un'entità ha cardinalità $(1, N)$, l'altra entità ha (x, N) , l'altra entità ancora ha cardinalità (x, N) (x può essere qualsiasi valore minimo) e un attributo è presente nella relazione.

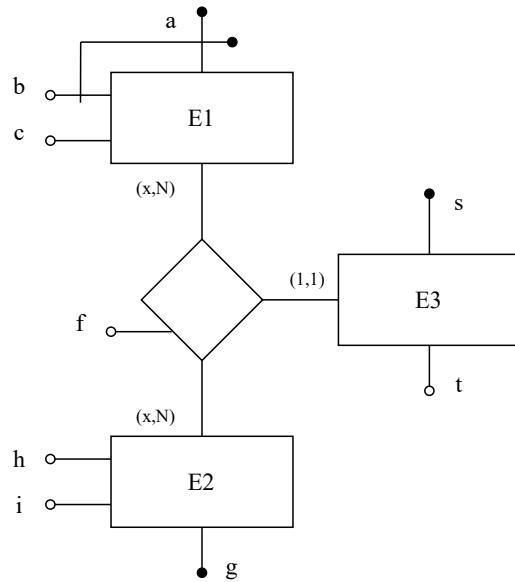
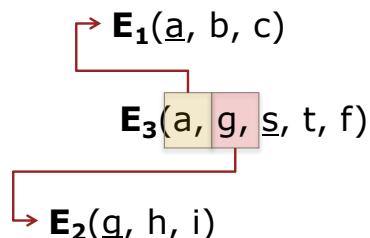


Figura 23: Modello E-R di una relazione molti a molti con relazione ternaria.

Relativo modello relazionale:



5 Algebra relazionale

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

- **Insiemistici**

- *Unione*
- *Intersezione*
- *Differenza*

- **Specifici**

- *Ridenominazione*
- *Selezione*
- *Proiezione*

- **Più importante join**

- *Join naturale*
- *Prodotto*
- *Cartesiano*
- *Semijoin*
- *Theta-join*

In altre parole, l'algebra relazione è un insieme di **operatori su relazioni**. Dato che le relazioni sono insiemi, ha senso definire su di esse gli operatori insiemistici come l'unione, la differenza e l'intersezione

5.1 Insiemistici

5.1.1 Unione

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

La **cardinalità**, ovvero il numero di tuple contenute nella relazione del risultato:

$$\max(|r_1|, |r_2|) \leq |r_1 \cup r_2| \leq |r_1| + |r_2|$$

Esempio

Laureati

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

Dirigenti

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

Laureati \cup Dirigenti

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

5.1.2 Intersezione

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

La **cardinalità**, ovvero il numero di tuple contenute nella relazione del risultato:

$$0 \leq |r_1 \cap r_2| \leq \min(|r_1|, |r_2|)$$

Esempio

Laureati

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

Dirigenti

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

Laureati \cap Dirigenti

Matricola	Cognome	Età
7432	Neri	39
9824	Verdi	38

5.1.3 Differenza

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

La **cardinalità**, ovvero il numero di tuple contenute nella relazione del risultato:

$$0 \leq |r_1 - r_2| \leq |r_1|$$

Esempio

Laureati

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

Dirigenti

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

Laureati – Dirigenti

Matricola	Cognome	Età
7274	Rossi	37

5.2 Specifici

5.2.1 Ridenominazione

L'**obiettivo** di questo operatore è risolvere le limitazioni degli operatori insiemistici. Infatti, esso **adegua i nomi degli attributi**, a seconda delle necessità, in particolare alla fine di facilitare le operazioni insiemistiche. Ovviamente, la ridenominazione avviene solamente sugli attributi, il **contenuto** rimane **inalterato**.

Il simbolo che lo rappresenta è la lettera greca rho ρ . **Al pedice viene inserita la ridenominazione, a destra il nome dell'attributo da rinominare e a sinistra il nuovo nome dell'attributo.** In generale, sia r una relazione definita sull'insieme di attributi X e sia Y un (altro) insieme di attributi con la stessa cardinalità. Inoltre, siano $A_1 A_2 \dots A_k$ e $B_1 B_2 \dots B_k$ rispettivamente un ordinamento per gli attributi in X e un ordinamento per quelli in Y . Allora la ridenominazione:

$$\rho_{B_1 B_2 \dots B_k \leftarrow A_1 A_2 \dots A_k} (r)$$

Contiene una tupla t' per ciascuna tupla t in r , definita come segue: t' è una tupla su Y e $t'[B_i] = t[A_i]$, per $i = 1, \dots, k$.

La definizione conferma che ciò che cambia sono i nomi degli attributi, mentre i valori rimangono inalterati e vengono associati ai nuovi attributi.

Esempio

Impiegati

Cognome	Agenzia	Stipendio
Rossi	Roma	45
Neri	Milano	53

Operai

Cognome	Fabbrica	Salario
Verdi	Latina	33
Bruni	Monza	32

Risultato (vedi sotto)

Cognome	Sede	Retribuzione
Rossi	Roma	45
Neri	Milano	53
Verdi	Latina	33
Bruni	Monza	32

Il risultato è ottenuto con la seguente operazione:

$$\rho_{\text{Sede}, \text{Retribuzione} \leftarrow \text{Agenzia}, \text{Stipendio}} (\text{Impiegati}) \cup \rho_{\text{Sede}, \text{Retribuzione} \leftarrow \text{Fabbrica}, \text{Salario}} (\text{Operai})$$

5.2.2 Selezione

La selezione produce una porzione dell'operando. Più precisamente, la **selezione** produce un sottoinsieme delle tuple su tutti gli attributi. Il **risultato** contiene le tuple dell'operando che soddisfano la condizione di selezione. Quest'ultima viene indicata nel pedice della notazione della selezione, ovvero sigma σ . Inoltre, le condizioni possono prevedere confronti fra attributi e confronti fra attributi e costanti, e possono essere complesse, ottenute combinando condizioni semplici con i connettivi logici \vee (or), \wedge (and) e \neg (not).

In generale, data una relazione $r(X)$, una *forma proposizionale* F su X è una formula ottenuta combinando, con i connettivi \wedge , \vee , \neg , condizioni atomiche del tipo $A\theta B$ o $A\theta c$, dove:

- θ è un **operatore di confronto**, il quale può essere:

$- =$
 $- \neq$
 $- >$
 $- <$
 $- \geq$
 $- \leq$

- A e B sono **attributi** in X sui cui valori il confronto θ abbia senso;
- c è una **costante** “compatibile” con il dominio di A (cioè tale che il confronto θ sia definito).

Data una formula F e una tupla t , è definito un valore di verità (cioè “vero” o “falso”) per F su t :

- $A\theta B$ è vera su t se $t[A]$ è in relazione θ con $t[B]$, altrimenti è falsa (per esempio, $A = B$ è vera su t se e solo se $t[A] = t[B]$);
- $A\theta c$ è vera su t se $t[A]$ è in relazione θ con c , altrimenti è falsa;
- $F_1 \vee F_2, F_1 \wedge F_2$ e $\neg F_1$ hanno l'usuale significato.

La **definizione**, in altre parole, è: la **selezione** $\sigma_F(r)$, in cui r è una relazione e F una formula proposizionale, produce una relazione sugli stessi attributi di r che contiene le tuple di r su cui F è vera.

Esempio 1

Impiegati

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

Risultato (vedi sotto)

Cognome	Nome	Età	Stipendio
Verdi	Nico	36	4.500,00

Il risultato è ottenuto con la seguente operazione:

$$\sigma_{\text{Eta} > 30 \wedge \text{Stipendio} > 4.000,00} (\text{Impiegati})$$

Esempio 2

Cittadini

Cognome	Nome	CittàDiNascita	Residenza
Rossi	Mario	Roma	Milano
Neri	Luca	Roma	Roma
Verdi	Nico	Firenze	Firenze
Rossi	Marco	Napoli	Firenze

Risultato (vedi sotto)

Cognome	Nome	CittàDiNascita	Residenza
Neri	Luca	Roma	Roma
Verdi	Nico	Firenze	Firenze

Il risultato è ottenuto con la seguente operazione:

$$\sigma_{\text{CittàDiNascita} = \text{Residenza}} (\text{Cittadini})$$

5.2.3 Proiezione

La **proiezione** dà un risultato cui contribuiscono tutte le tuple, ma su un sottoinsieme degli attributi.

Formalmente, dati una relazione $r(X)$ e un sottoinsieme Y di X , la **proiezione** di r su Y (indicata con $\pi_Y(r)$) è l'insieme di tuple su Y ottenute dalle tuple di r considerando solo i valori su Y :

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

Dagli esempi è chiaro che la proiezione permette di decomporre verticalmente le relazioni.

Esempio 1

In questo caso, il risultato della proiezione contiene tante tuple quante l'operando, definite però solo su parte degli attributi.

Impiegati

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

Risultato (vedi sotto)

Cognome	Nome
Rossi	Mario
Neri	Luca
Verdi	Mario
Rossi	Marco

Il risultato è ottenuto con la seguente operazione:

$$\pi_{\text{Cognome}, \text{Nome}}(\text{Impiegati})$$

Esempio 2

In questo caso, il risultato contiene un numero di tuple inferiore rispetto a quelle dell'operando, perché alcune tuple, avendo uguali valori su tutti gli attributi della proiezione, danno lo stesso contributo alla proiezione stessa. Essendo le relazioni definite come insieme, non possono, per definizione, in esse comparire più tuple uguale fra loro: i contributi “collassano” in una sola tupla.

Impiegati

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

Risultato (vedi sotto)

Reparto	Capo
Vendite	Gatti
Personale	Lupi

Il risultato è ottenuto con la seguente operazione:

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

5.3 Join

L'operatore di *join* consente di correlare dati contenuti in relazioni diverse, confrontando i valori contenuti in esse e utilizzando quindi la caratteristica fondamentale del modello, ovvero quella di essere basta su valori.

Formalmente, due relazione r_1 e r_2 di schema X_1 e X_2 rispettivamente, gli operatori di *join* generano tuple t nella relazione risultato a partire dalle coppie di tuple $(t_1, t_2) \in r_1 \times r_2$ che soddisfano una certa condizione (chiamata **predicato di join**).

5.3.1 Join naturale

Il **join naturale** è un operatore che correla dati in relazioni diverse, sulla base di valori uguali in attributi con lo stesso nome (si veda l'esempio come chiarificatore). Questa operazione viene denotata con il simbolo \bowtie .

Il **risultato** dell'operatore è 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. Nel primo esempio sotto, la prima tupla del *join* deriva dalla combinazione della prima tupla della relazione r_1 e dalla seconda tupla della relazione r_2 .

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

$$r_1 \bowtie r_2 = \{t \text{ su } X_1 X_2 \mid t[X_1] \in r_1, t[X_2] \in r_2\}$$

Si noti che è molto frequente eseguire *join* sulla base di valori della chiave di una delle relazioni coinvolte, esplicitando i riferimenti fra tuple che sono realizzati per mezzo di valori soprattutto valori di chiavi. Osservando l'esempio 2, si vede che ciascuna delle tuple di **Infrazioni** è stata combinata con una e una sola delle tuple di **Auto**:

- Una sola perché **Prov** e **Numero** formano una chiave di **Auto**;
- Almeno una perché è definito il vincolo di integrità referenziale fra **Prov** e **Numero** in **Infrazioni** e (la chiave primaria di) **Auto**

Dunque, il *join* ha esattamente tante tuple quante la relazione **Infrazioni**.

Esempio 1

r_1		r_2	
Impiegato	Reparto	Reparto	Capo
Rossi	vendite	produzione	Mori
Neri	produzione	vendite	Bruni
Bianchi	produzione		

Risultato (vedi sotto)

Impiegato	Reparto
Rossi	vendite
Neri	produzione
Bianchi	produzione

Il risultato è ottenuto con la seguente operazione:

$$r_1 \bowtie r_2$$

Esempio 2

Infrazioni

Codice	Data	Agente	Articolo	Stato	Numero
143256	25/10/2017	567	44	I	AB 234 ZK
987554	26/10/2017	456	34	I	AB 234 ZK
987557	26/10/2017	456	34	I	CB 123 AA
630876	15/10/2017	456	53	F	CB 123 AA
539856	12/10/2017	567	44	F	CB 123 AA

Auto

Stato	Numero	Proprietario	Indirizzo
I	CB 123 AA	Verdi Piero	Via Tigli
I	DE 834 ZZ	Verdi Piero	Via Tigli
I	AB 234 ZK	Bini Luca	Via Aceri
F	CB 123 AA	Beau Marcel	Rue Louis

Risultato (vedi sotto)

Codice	Data	Agente	Articolo	Stato	Numero	Proprietario	Indirizzo
143256	25/10/2017	567	44	I	AB 234 ZK	Bini Luca	Via Aceri
987554	26/10/2017	456	34	I	AB 234 ZK	Bini Luca	Via Aceri
987557	26/10/2017	456	34	I	CB 123 AA	Verdi Piero	Via Tigli
630876	15/10/2017	456	53	F	CB 123 AA	Beau Marcel	Rue Louis
539856	12/10/2017	567	44	F	CB 123 AA	Beau Marcel	Rue Louis

Il risultato è ottenuto con la seguente operazione:

$$\text{Infrazioni} \bowtie \text{Auto}$$

5.3.2 Join completi e incompleti

Nell'esempio 1 nel paragrafo del *join naturale*, si può dire che ciascuna tupla di ciascuno degli operandi contribuisce almeno una tupla del risultato (in questo caso il *join* viene detto **completo**): per ogni tupla t_1 di r_1 , esiste una tupla t in $r_1 \bowtie r_2$ tale che $t[X_1] = t_1$ (e analogamente per r_2).

L'esempio 1 a fine paragrafo, mostra un *join* in cui alcune tuple degli operandi, in particolare la prima di r_1 e la seconda di r_2 , non contribuiscono al risultato, perché l'altra relazione non contiene tuple con gli stessi valori sull'attributo comune. In questo caso il *join* viene detto **dangling**, ovvero **dondolante**.

Infine, come caso limite, è ovviamente possibile che nessuna delle tuple degli operandi sia combinabile, e allora il risultato del *join* è la **relazione vuota** (esempio 2 a fine paragrafo). All'estremo opposto, è possibile che ciascuna delle tuple di ciascuno degli operandi sia combinabile con tutte dell'altro, come mostrato nell'ultimo esempio del paragrafo, e in questo caso il risultato ha un numero di tuple pari al prodotto delle cardinalità degli operandi e cioè $|r_1| \times |r_2|$ (dove $|r|$ indica la cardinalità della relazione r).

Alcune osservazioni finali:

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

Esempio 1

r₁			
Impiegato	Reparto	Reparto	Capo
Rossi	vendite	produzione	Mori
Neri	produzione	acquisti	Bruni
Bianchi	produzione		

Risultato (vedi sotto)

Impiegato	Reparto	Capo
Neri	produzione	Mori
Bianchi	produzione	Mori

Il risultato è ottenuto con la seguente operazione:

$$r_1 \bowtie r_2$$

Esempio 2

r₁			
Impiegato	Reparto	Reparto	Capo
Rossi	vendite	concorsi	Mori
Neri	produzione	acquisti	Bruni
Bianchi	produzione		

Risultato (vedi sotto)

Impiegato	Reparto	Capo

Il risultato è ottenuto con la seguente operazione:

$$r_1 \bowtie r_2$$

Esempio 3

r₁			
Impiegato	Progetto	Progetto	Capo
Rossi	A		Mori
Neri	A		Bruni
Bianchi	A		

Risultato (vedi sotto)

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

Il risultato è ottenuto con la seguente operazione:

$$r_1 \bowtie r_2$$

5.3.3 Theta-join ed equi-join

Osservando l'esempio 1 a fine paragrafo, è possibile notare come un prodotto cartesiano ha di solito ben poca utilità, in quanto concatena tuple non necessariamente correlate dal punto di vista semantico. Infatti, il **prodotto cartesiano viene spesso seguito da una selezione**, la quale centra l'attenzione su tuple correlate secondo le esigenze.

Per esempio, nell'esempio 2 a fine paragrafo, sulle relazioni *Impiegati* e *Progetti* ha senso definire un prodotto cartesiano seguito dalla selezione che mantiene solo le tuple con valori uguali sull'attributo *Progetto* di *Impiegati* e su *Codice* di *Progetti*.

Per questo motivo, viene definito un operatore derivato (cioè per mezzo di altri operatori), chiamato **theta-join**. Esso è considerato come un **prodotto cartesiano seguito da una selezione**, nel modo seguente:

$$r_1 \underset{F}{\bowtie} r_2 = \sigma_F(r_1 \bowtie r_2)$$

Attenzione! Gli schemi r_1 e r_2 devono essere **disgiunti**, cioè:

$$r_1 \cap r_2 = \emptyset$$

Per esempio, nell'esempio 2 a fine paragrafo, la relazione può essere ottenuta per mezzo del *theta-join*:

$$\begin{array}{c} \text{Impiegati} \quad \bowtie \quad \text{Progetti} \\ \text{Progetto} = \text{Codice} \end{array}$$

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

Infine, il **join naturale è possibile simularlo** tramite la ridenominazione, l'*equi-join* e la proiezione. Date due relazioni $r_1(ABC)$ e $r_2(BCD)$, il join naturale di r_1 e r_2 può essere espresso per mezzo degli altri operatori, in tre passi:

1. **Ridenominando** gli attributi così da ottenere relazioni su schemi disgiunti:

$$\rho_{B'C' \leftarrow BC}(r_2)$$

2. Effettuando l'**equi-join**, con condizioni di uguaglianza sugli attributi corrispondenti:

$$r_1 \underset{B=B' \wedge C=C'}{\bowtie} \rho_{B'C' \leftarrow BC}(r_2)$$

3. Concludendo con una **proiezione** che elimina gli attributi "doppioni", che presentano valori identici a quelli di altri attributi:

$$\pi_{ABCD} \left(r_1 \underset{B=B' \wedge C=C'}{\bowtie} \rho_{B'C' \leftarrow BC}(r_2) \right)$$

Esempio 1

Impiegati

Impiegato	Progetto
Rossi	A
Neri	A
Neri	B

Progetti

Codice	Nome
A	Venere
B	Marte

Risultato (vedi sotto)

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

Il risultato è ottenuto con la seguente operazione:

Impiegati \bowtie Progetti

Esempio 2

Impiegati

Impiegato	Progetto
Rossi	A
Neri	A
Neri	B

Progetti

Codice	Nome
A	Venere
B	Marte

Risultato (vedi sotto)

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

Il risultato è ottenuto con la seguente operazione:

$$\sigma_{\text{Progetto}=\text{Codice}} (\text{Impiegati} \bowtie \text{Progetti})$$

5.4 Algebra con valori nulli

È necessario introdurre la possibilità di avere dei valori nulli nell'algebra relazionale. Un **valore nullo** (*unknown*) viene rappresentato con il simbolo U e un **predicato prende tale valore quando almeno uno dei termini del confronto è sconosciuto**. Considerando l'esempio a fine paragrafo, data la seguente selezione:

$$\sigma_{\text{Eta} > 30} (\text{Persone})$$

La prima tupla certamente appartiene al risultato (appartenenza **vera**), la seconda certamente non appartiene (appartenenza **falsa**), la terza forse appartiene e forse no (appartenenza **sconosciuta**).

Le **tavola di verità** dei connettivi *not*, *and*, *or* tenendo conto del nuovo valore logico, si estendono nel seguente modo: Il valore nullo rappresenta un valore di

<i>not</i>	<i>and</i>	<i>or</i>
V	V U F	V V V V
U	U U F	V U U
F	F F F	V U F

verità intermedio tra vero e falso, e il significato dei tre connettivi in questo contesto diventa il seguente:

- Il ***not*** è vero solo se il valore di partenza è falso;
- L'***and*** è vero solo se tutti i termini sono veri;
- L'***or*** è vero se almeno uno dei termini è vero.

Considerando ancora l'esempio a fine paragrafo, data la seguente espressione:

$$\sigma_{\text{Eta} > 30} (\text{Persone}) \cup \sigma_{\text{Eta} \leq 30} (\text{Persone})$$

Logicamente si potrebbe pensare che il risultato sia esattamente la relazione Persone. Questo è **sbagliato** poiché la terza tupla, quella con il valore NULL, ha un'appartenenza sconosciuta a ciascuna sottoespressione e dunque anche all'unione. Questo discorso vale anche per la seguente espressione:

$$\sigma_{\text{Eta} > 30 \vee \text{Eta} \leq 30} (\text{Persone})$$

In cui la disgiunzione viene valutata secondo la logica a tre valori.

Da questo problema nasce l'esigenza di esplicitare due nuove condizioni: **IS NULL** e **IS NOT NULL**. Il loro significato:

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

Considerando l'esempio a fine paragrafo, data l'espressione:

$$\sigma_{\text{Eta} > 30 \vee \text{Eta IS NULL}} (\text{Persone})$$

Si ottiene le persone che hanno o potrebbero avere più di trent'anni (quindi età nota e maggiore di 30 oppure non nota). Invece, per ottenere tutte le tuple:

$$\sigma_{\text{Eta} > 30 \vee \text{Eta} \leq 30 \vee \text{Eta IS NULL}} (\text{Persone})$$

Esempio

Persone

Nome	Età	Reddito
Aldo	35	15
Andrea	27	21
Maria	NULL	42

5.5 Ottimizzare ed equivalenza di espressioni algebriche

5.5.1 Definizioni

Ogni espressione (*query*) ricevuta dal DBMS è soggetta ad un processo di elaborazione. Esiste una parte software ad un livello più basso che implementa l'**ottimizzatore**. Esso ha l'obiettivo di generare un'espressione equivalente all'originale ma con un costo minore. Per **costo** si intende la dimensione dei risultati intermedi.

Da questa necessità, si crea l'**equivalenza delle espressioni algebriche**, cioè espressioni fra loro equivalenti che producono quindi lo stesso risultato.

Esistono **due tipi** di equivalenza:

- **Equivalenza assoluta.** Non viene esplicitato nessuno schema, quindi è completamente indipendente (assoluta).

Più formalmente:

$$E_1 \equiv E_2 \text{ se } E_1 \equiv_{\mathbf{R}} E_2 \text{ per ogni schema } \mathbf{R}$$

Un esempio:

$$\pi_{AB}(\sigma_{A>0}(R)) \equiv \sigma_{A>0}(\pi_{AB}(R))$$

- **Equivalenza dipendente dallo schema.** Viene esplicitato lo schema e dunque l'equivalenza è vera se e solo se l'intersezione degli insiemi è uguale.

Più formalmente:

$$E_1 \equiv_{\mathbf{R}} E_2 \text{ se } E_1(\mathbf{r}) = E_2(\mathbf{r}) \text{ per ogni istanza } \mathbf{r} \text{ di } \mathbf{R}$$

Un esempio:

$$\pi_{AB}(R_1) \bowtie \pi_{AC}(R_2) \equiv_{\mathbf{R}} \pi_{ABC}(R_1 \bowtie R_2)$$

L'equivalenza è valida se e solo se nello schema **R** l'intersezione fra gli insiemi di attributi di R_1 e R_2 è pari ad A .

5.5.2 Trasformazioni di equivalenza

Nel contesto delle ottimizzazioni, vengono spesso utilizzate le **trasformazioni di equivalenza**, ovvero operazioni che **sostituiscono un'espressione con un'altra a essa equivalente**. In particolare, risultano interessanti le trasformazioni che riducono le dimensioni dei risultati intermedi e quelle che preparano un'espressione all'applicazione di una trasformazione che riduce le dimensioni dei risultati intermedi.

- I. **Atomizzazione delle selezioni**: una selezione σ congiuntiva¹ può essere sostituita da una cascata di selezioni atomiche:

$$\sigma_{F_1 \wedge F_2}(E) \equiv \sigma_{F_1}(\sigma_{F_2}(E))$$

In cui E è una qualunque espressione.

- II. **Idempotenza delle proiezioni**: una proiezione π può essere trasformata in una cascata di proiezioni che “eliminano” i vari attributi in fasi successive:

$$\pi_X(E) \equiv \pi_X(\pi_{XY}(E))$$

Se E è definita su un insieme di attributi che contiene Y (oltre a X).

- III. **Pushing selections down (Anticipazione della selezione rispetto al join)**:

$$\sigma_F(E_1 \bowtie E_2) \equiv E_1 \bowtie \sigma_F(E_2)$$

Se la condizione F fa riferimento solo ad attributi nella sottoespressione E_2 .

- IV. **Pushing projections down (Anticipazione della proiezione rispetto al join)**: siano E_1 ed E_2 definite rispettivamente su X_1 e X_2 . Allora, se $Y_2 \subseteq X_2$ e $Y_2 \supseteq (X_1 \cap X_2)$, ovvero se gli attributi in $X_2 - Y_2$ non sono coinvolti nel join, vale:

$$\pi_{X_1 Y_2}(E_1 \bowtie E_2) \equiv E_1 \bowtie \pi_{Y_2}(E_2)$$

Combinando questa regola con quella della idempotenza delle proiezioni, si può ottenere:

$$\pi_Y\left(E_1 \underset{F}{\bowtie} E_2\right) \equiv \pi_Y\left(\pi_{Y_1}(E_1) \underset{F}{\bowtie} \pi_{Y_2}(E_2)\right)$$

Si indica con X_1 e X_2 gli attributi di E_1 ed E_2 rispettivamente, con J_1 e J_2 i rispettivi sottoinsiemi coinvolti nella condizione F di join:

$$\begin{aligned} Y_1 &= (X_1 \cap Y) \cup J_1 \\ Y_2 &= (X_2 \cap Y) \cup J_2 \end{aligned}$$

In altre parole, è possibile **eliminare subito da ciascuna relazione gli attributi che non compaiono nel risultato finale e non sono coinvolti nel join**.

¹Ovvero due espressioni connesse da un and, or, not logico

V. **Inglobamento di una selezione in un prodotto cartesiano a formare un join:**

$$\sigma_F (E_1 \bowtie E_2) \equiv E_1 \underset{F}{\bowtie} E_2$$

Indicando X_1 e X_2 i rispettivi attributi di E_1, E_2 , allora $X_1 \cap X_2 = \emptyset$.

Queste trasformate sono quelle più importanti.

Esistono altre trasformate non meno importanti ma di minore rilievo rispetto alle prime che sono considerate fondamentali.²

VI. **Distributività della selezione rispetto all'unione:**

$$\sigma_F (E_1 \cup E_2) \equiv \sigma_F (E_1) \cup \sigma_F (E_2)$$

VII. **Distributività della selezione rispetto alla differenza:**

$$\sigma_F (E_1 - E_2) \equiv \sigma_F (E_1) - \sigma_F (E_2)$$

VIII. **Distributività della proiezione rispetto all'unione:**

$$\pi_X (E_1 \cup E_2) \equiv \pi_X (E_1) \cup \pi_X (E_2)$$

Un altro gruppo minore di trasformate è quello basato sull'interazione fra gli operatori insiemistici e le selezioni complesse:

IX. $\sigma_{F_1 \vee F_2} (R) \equiv \sigma_{F_1} (R) \cup \sigma_{F_2} (R)$

X. $\sigma_{F_1 \wedge F_2} (R) \equiv \sigma_{F_1} (R) \cap \sigma_{F_2} (R) \equiv \sigma_{F_1} (R) \bowtie \sigma_{F_2} (R)$

XI. $\sigma_{F_1 \wedge \neg F_2} (R) \equiv \sigma_{F_1} (R) - \sigma_{F_2} (R)$

Infine, la **proprietà distributiva del join rispetto all'unione:**

XII. $E \bowtie (E_1 \cup E_2) \equiv (E \bowtie E_1) \cup (E \bowtie E_2)$

²All'esame non verranno chieste le trasformate minori!

Esempio

Si vede un esempio per chiarire le trasformazioni più importanti. Data la seguente base di dati:

Impiegati				Supervisione	
Matricola	Nome	Età	Stipendio	Capo	Impiegato
101	Mario Rossi	34	40	210	101
103	Mario Bianchi	23	35	210	103
104	Luigi Neri	38	61	210	104
105	Nico Bini	44	38	231	105
210	Marco Celli	49	60	301	210
231	Siro Bisi	50	60	301	231
252	Nico Bini	44	70	375	252
301	Sergio Rossi	34	70		
375	Mario Rossi	50	65		

Si vuole trovare il numero di matricola dei capi di impiegati con meno di trenta anni. Immediatamente si potrebbe pensare di eseguire un join sulle due tabelle, specificare la condizione di selezione e infine proiettare l'unica colonna d'interesse:

$$\pi_{\text{Capo}} (\sigma_{\text{Impiegato}=\text{Matricola} \wedge \text{Età} < 30} (\text{Impiegati} \bowtie \text{Supervisione}))$$

È evidente che l'espressione risulta qualitativamente bassa poiché per calcolare pochi valori, in questo caso uno, viene effettuato un prodotto cartesiano che produce un risultato notevole. Quindi, si esegue l'ottimizzazione:

1. (Regola 1) Atomizzazione delle selezioni. Si elimina la coniugazione logica and:

$$\pi_{\text{Capo}} (\sigma_{\text{Impiegato}=\text{Matricola}} (\sigma_{\text{Età} < 30} (\text{Impiegati} \bowtie \text{Supervisione})))$$

2. (Regola 3) Anticipazione della selezione rispetto al join e (Regola 5) inglobamento di una selezione in un prodotto cartesiano a formare un join. Si fonde la prima selezione ($\sigma_{\text{Impiegato}=\text{Matricola}}$) con il prodotto cartesiano formando un join con condizione e successivamente si anticipa la seconda selezione ($\sigma_{\text{Età} < 30}$) rispetto al join:

$$\pi_{\text{Capo}} \left(\sigma_{\text{Età} < 30} (\text{Impiegati}) \underset{\text{Impiegato}=\text{Matricola}}{\bowtie} \text{Supervisione} \right)$$

3. (Regola 4) Anticipazione della proiezione rispetto al join. Si elimina dal primo argomento del join anche gli attributi non necessari:

$$\pi_{\text{Capo}} \left(\pi_{\text{Matricola}} (\sigma_{\text{Età} < 30} (\text{Impiegati})) \underset{\text{Impiegato}=\text{Matricola}}{\bowtie} \text{Supervisione} \right)$$

6 Calcolo relazionale

Il **calcolo relazionale** fa riferimento ad una famiglia di linguaggi di interrogazione, basati sul calcolo dei predicati del primo ordine, che hanno la caratteristica di essere **dichiarativi**, cioè di **specificare le proprietà del risultato delle interrogazioni, anziché la procedura seguita per generarlo**.

Al contrario l'algebra relazionale è un **linguaggio procedurale** poiché le sue espressioni specificano passo passo la costruzione del risultato.

Esistono diversi **tipi** di calcolo relazionale:

- **Calcolo relazione su domini.** Presenta in modo naturale le caratteristiche originali dei linguaggi del calcolo relazionale.
- **Calcolo su tuple con dichiarazioni di range.** Metodo adottato durante questo corso, costituisce la base per molti costrutti disponibili per le interrogazioni nel linguaggio SQL.

Prima di presentare le varie caratteristiche, si tiene a precisare che rispetto al modello relazionale (paragrafo 2.3), il calcolo su tuple con dichiarazioni di range utilizza una **notazione non posizionale**.

6.1 Calcolo su tuple con dichiarazioni di range

Il calcolo su tuple con dichiarazioni di range presenta la seguente forma:

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

In cui le variabili hanno diversi significati:

- La variabile T indica la **target list**, ovvero la lista degli obiettivi dell'interrogazione. I suoi elementi hanno la seguente forma $Y : x.Z$ con x variabile e Y e Z sequenze di attributi (di pari lunghezza). Chiaramente, gli attributi in Z devono comparire nello schema della relazione che costituisce il *range* di x .

Una notazione alternativa per abbreviare $X : x.X$ è $x.*$;

- La variabile L indica la **range list**, ovvero la lista contenente le variabili libere della formula f con i relativi *range*. I suoi elementi hanno la seguente forma $x(R)$ con x variabile e R nome di relazione;
- La variabile f indica una **formula**, la quale può essere di tre tipi:

– Formula con atomi del tipo:

- * $x.A\theta c$, si confronta il valore di x sull'attributo A con la costante c ;
- * $x_1.A_1\theta x_2.A_2$, si confronta il valore di x_1 su A_1 con quello di x_2 su A_2 .

– Formula con connettivi (\wedge, \vee, \neg);

– Formula con quantificatori che associano i range alle relative variabili:

$$\exists x(R)(f) \quad \forall x(R)(f)$$

Intuitivamente, $\exists x(R)(f)$ significa “esiste nella relazione R una tupla x che soddisfa la formula f ”.

6.2 Unione, intersezione e differenza

Purtroppo, il calcolo su tuple con dichiarazioni di range non permette di esprimere tutte le interrogazioni che possono essere formulate in algebra relazionale.

In particolare, le interrogazioni i cui risultati possono provenire indifferentemente da due o più relazioni (in algebra si realizza con un'unione) non possono essere espresse in questa versione del calcolo.

Infatti, i risultati sono costituiti a partire da tutte le variabili libere, i cui range sono definiti nella target list, e ogni variabile ha come range una sola relazione.

Per esempio, si consideri un'unione di due relazioni sugli stessi attributi: $R_1(AB)$ e $R_2(AB)$. Se l'espressione avesse due variabili libere, allora ogni tupla del risultato dovrebbe corrispondere a una tupla di ciascuna delle relazioni, il che non è necessario, poiché l'unione richiede alle tuple nel risultato di comparire in almeno uno degli operandi, non necessariamente in entrambi. Viceversa, se l'espressione avesse una sola variabile libera, questa dovrebbe far riferimento a una sola delle relazioni, senza acquisire tuple dell'altra per il risultato.

Nonostante l'**operatore di unione non sia rappresentabile**, gli operatori di intersezione e differenza risultano esprimibili:

- L'**intersezione** richiede che le tuple siano in entrambi gli operandi, ovvero richiede l'esistenza di una tupla uguale nell'altra relazione. Per esempio:

$$\pi_{BC}(R_1) \cap \pi_{BC}(R_2)$$

Si esprime come:

$$\{x_1.BC \mid x_1(R_1) \mid \exists x_2(R_2)(x_1.B = x_2.B \wedge x_1.C = x_2.C)\}$$

- La **differenza** produce le tuple di un operando non contenute nell'altro e può essere specificata richiedendo le tuple del minuendo che non compaiono nel sottraendo. Per esempio:

$$\pi_{BC}(R_1) - \pi_{BC}(R_2)$$

Si esprime come:

$$\{x_1.BC \mid x_1(R_1) \mid \neg \exists x_2(R_2)(x_1.B = x_2.B \wedge x_1.C = x_2.C)\}$$

6.3 Esempi

Tutte le interrogazioni che verranno affrontate riguardano la seguente basi di dati (già vista nei precedenti paragrafi 5.5.2):

Impiegati				Supervisione	
Matricola	Nome	Età	Stipendio	Capo	Impiegato
101	Mario Rossi	34	40	210	101
103	Mario Bianchi	23	35	210	103
104	Luigi Neri	38	61	210	104
105	Nico Bini	44	38	231	105
210	Marco Celli	49	60	301	210
231	Siro Bisi	50	60	301	231
252	Nico Bini	44	70	375	252
301	Sergio Rossi	34	70		
375	Mario Rossi	50	65		

La prima interrogazione è la seguente:

1. Si richiede la matricola, il nome, l'età e lo stipendio degli impiegati che guadagnano più di 40 mila euro:

$$\{i.* \mid i(\text{Impiegati}) \mid i.\text{Stipendio} > 40\}$$

Per produrre meno risultati è sufficiente modificare la *target list*:

2. Si richiede la matricola, il nome e l'età degli impiegati che guadagnano più di 40 mila euro:

$$\{i.(\text{Matricola}, \text{Nome}, \text{Età}) \mid i(\text{Impiegati}) \mid i.\text{Stipendio} > 40\}$$

Per eseguire interrogazioni che coinvolgono più relazioni, si modifica la *range list* usando più variabili. Si noti la prima condizione atomica che corrisponde a quella del join (e l'altra per la selezione):

3. Trovare le matricole dei capi degli impiegati che guadagnano più di 40 mila euro:

$$\{s.(\text{Capo}) \mid i(\text{Impiegati}), s(\text{Supervisione}) \mid i.\text{Matricola} = s.\text{Impiegato} \wedge i.\text{Stipendio} > 40\}$$

Nel caso di join di una relazione con se stessa si ha più variabili aventi la stessa relazione come range:

4. Trovare il nome e stipendio dei capi degli impiegati che guadagnano più di 40 mila euro:

$$\{\text{NomeC}, \text{StipC} : i'.(\text{Nome}, \text{Stipendio}) \mid i'(\text{Impiegati}), s(\text{Supervisione}), i(\text{Impiegati}) \mid i'.\text{Matricola} = s.\text{Capo} \wedge s.\text{Impiegato} = i.\text{Matricola} \wedge i.\text{Stipendio} > 40\}$$

5. Trovare gli impiegati che guadagnano più del rispettivo capo, mostrando matricola, nome e stipendio di ciascuno di essi e del capo:

$$\{i.(\text{Nome}, \text{Matr}, \text{Stip}), \text{NomeC}, \text{MatrC}, \text{StipC} : i'.(\text{Nome}, \text{Matr}, \text{Stip}) \mid i'(\text{Impiegati}), s(\text{Supervisione}), i'(\text{Impiegati}) \mid i.\text{Matr} = s.\text{Impiegato} \wedge s.\text{Capo} = i'.\text{Matr} \wedge i.\text{Stipendio} > i'.\text{Stipendio}\}$$

Le interrogazioni con i quantificatori (\exists, \forall) mostrano appieno la maggiore sinteticità e praticità del calcolo su tuple con dichiarazioni di range:

6. Trovare la matricola e il nome dei capi i cui impiegati guadagnano tutti più di 40 mila:

- Quantificatore universale:

$$\{i.(\text{Matricola}, \text{Nome}) \mid i(\text{Impiegati}), s(\text{Supervisione}) \mid i.\text{Matr} = s.\text{Capo} \wedge \forall i'(\text{Impiegati}) (\forall s'(\text{Supervisione}) (\neg(s.\text{Capo} = s'.\text{Capo} \wedge s'.\text{Impiegato} = i'.\text{Matr}) \vee i'.\text{Stipendio} > 40)))\}$$

- Quantificatore esistenziale:

$$\{i.(\text{Matricola}, \text{Nome}) \mid i(\text{Impiegati}), s(\text{Supervisione}) \mid i.\text{Matr} = s.\text{Capo} \wedge \neg(\exists i'(\text{Impiegati}) (\exists s'(\text{Supervisione}) (s.\text{Capo} = s'.\text{Capo} \wedge s'.\text{Impiegato} = i'.\text{Matr} \wedge i'.\text{Stipendio} \leq 40))))\}$$

6.4 Esercizi

6.4.1 Aula, insegnamento, docente e lezione

Testo

Dato il seguente schema relazionale:

- **AULA**(NomeAula, Capienza, Edificio);
- **INSEGNAMENTO**(NomeIns, AnnoAcc, Docente);
- **DOCENTE**(Matricola, Nome, Cognome, Età, Ruolo: {ordinario, associato, ricercatore, esterno});
- **LEZIONE**(NomeIns, AnnoAcc, NomeAula, Giorno, Semestre, OraInizio, OraFine).

Si calcoli:

1. Trovare il nome e la capienza delle aule dove nel 2° semestre 2015/2016 il venerdì non si sono svolte lezioni.
2. Trovare i docenti che nel 1° semestre 2016/2017 hanno svolto almeno una lezione di durata maggiore di 180 minuti, riportando nel risultato il nome e il cognome del docente insieme alla durata della lezione (durata in minuti = (OraFine – OraInizio)).
3. Trovare per ogni lezione del 1° semestre 2016/2017 che si svolge il martedì prima delle 17.00 in aula A, la lezione immediatamente successiva nella stessa aula, riportando nel risultato per la prima lezione il nome dell'insegnamento, l'ora di inizio e l'ora di fine e per la lezione successiva solo il nome dell'insegnamento.

Soluzione 1

Trovare il nome e la capienza delle aule dove nel 2° semestre 2015/2016 il venerdì non si sono svolte lezioni.

La soluzione è la seguente:

$$Q = \{\text{Nome, Capienza} : A(\text{NomeAula}, \text{Capienza}) \mid A(\text{Aula}) \mid \neg \exists L(\text{Lezione}) (A(\text{NomeAula}) = L(\text{NomeAula}) \wedge L(\text{AnnoAcc}) = "2015/2016" \wedge L(\text{Semestre}) = "2^o" \wedge L(\text{Giorno}) = "Venerdì")\}$$

Il nome e la capienza delle aule si trovano nella tabella Aula, per cui nella *target list* andranno i suoi due campi e basta.

Nella *range list* è sufficiente l'aula, nonostante nella tabella lezione ci siano alcuni dati importanti.

Nel campo formula ci sono alcune condizioni. Per manifestare la negazione è necessario l'operatore logico *not*. Esso deve operare su un campo presente nella tabella Lezione, ma allo stesso tempo deve avere le caratteristiche richieste dal risultato. Quindi, si scrive che non esiste una *L* appartenente alla tabella Lezione tale per cui essa abbia:

- Il nome dell'aula (chiave) uguale sia nella tabella Aula che Lezione;
- L'anno accademico (tabella Lezione) uguale al valore 2015/2016;
- Il semestre (tabella Lezione) uguale al valore 2^o;
- Il giorno (tabella Lezione) uguale al valore Venerdì.

Quindi, in questo caso si richiede una Lezione che non è presente nella tabella Aula, cioè che non si sia svolta. Tuttavia, oltre a non essersi svolta, deve in un anno, semestre e giorno preciso.

Soluzione 2

Trovare i docenti che nel 1° semestre 2016/2017 hanno svolto almeno una lezione di durata maggiore di 180 minuti, riportando nel risultato il nome e il cognome del docente insieme alla durata della lezione (durata in minuti = (OraFine – OraInizio)).

La soluzione è la seguente:

$$\begin{aligned} Q = \{ & \text{Nome, Cognome : } D(\text{Nome, Cognome}), \\ & \text{Durata Lezione : } L(\text{OraFine}) - L(\text{OraInizio}) \quad | \\ & D(\text{Docente}), I(\text{Insegnamento}), L(\text{Lezione}) \quad | \\ & D(\text{Matricola}) = I(\text{Docente}) \wedge I(\text{NomeIns}) = L(\text{NomeIns}) \wedge \\ & I(\text{AnnoAcc}) = L(\text{AnnoAcc}) \wedge L(\text{AnnoAcc}) = "2016/2017" \wedge L(\text{Semestre}) = 1^{\circ} \wedge \\ & (L(\text{OraFine}) - L(\text{OraInizio})) \geq 180 \} \end{aligned}$$

Il risultato richiede il nome e il cognome del docente, quindi la *target list* avrà tali campi e la durata. Quest'ultima sarà specificata eseguendo la differenza tra l'ora di fine e l'ora di inizio.

Nella *range list* si specifica ovviamente la tabella del docente, dell'insegnamento e della lezione.

Nel campo formula ci sono alcune condizioni. È necessario collegare tutte le chiavi esterne, quindi le matricole del docente, il nome e l'anno accademico tra l'Insegnamento e la Lezione. Inoltre, le condizioni sul numero di semestre, anno accademico e durata delle lezioni, cioè ora di inizio e fine, sono specificate nella tabella Lezione. Quindi, le condizioni vengono espresse su di essa.

Soluzione 3

Trovare per ogni lezione del 1° semestre 2016/2017 che si svolge il martedì prima delle 17.00 in aula A, la lezione immediatamente successiva nella stessa aula, riportando nel risultato per la prima lezione il nome dell'insegnamento, l'ora di inizio e l'ora di fine e per la lezione successiva solo il nome dell'insegnamento.

La soluzione è la seguente:

$$\begin{aligned} Q = \{ & \text{Ins, Inizio, Fine : } L1.(\text{NomeIns}, \text{OraInizio}, \text{OraFine}), \\ & \text{InsSuccessivo : } L2.(\text{NomeIns}) \quad | \quad L1(\text{LEZIONE}), L2(\text{LEZIONE}) \quad | \\ & L1.\text{Semestre} = "1" \wedge L2.\text{Semestre} = "1" \wedge L1.\text{AnnoAcc} = "2016/2017" \wedge \\ & L2.\text{AnnoAcc} = "2016/2017" \wedge L1.\text{Giorno} = "\text{Martedì}" \wedge L2.\text{Giorno} = "\text{Martedì}" \wedge \\ & L1.\text{NomeAula} = "A" \wedge L2.\text{NomeAula} = "A" \wedge L1.\text{OraInizio} < 17:00 \wedge \\ & L1.\text{OraInizio} < L2.\text{OraInizio} \wedge \neg \exists L3(\text{LEZIONE}) (L3.\text{Semestre} = "1" \wedge \\ & L3.\text{AnnoAcc} = "2016/2017" \wedge L3.\text{Giorno} = "\text{Martedì}" \wedge L3.\text{NomeAula} = "A" \wedge \\ & L1.\text{OraInizio} < L3.\text{OraInizio} \wedge L3.\text{OraInizio} < L2.\text{OraInizio}) \} \end{aligned}$$

Il risultato richiede il nome, l'ora di inizio e l'ora di fine dell'insegnamento della prima lezione. Inoltre, anche il nome dell'insegnamento della lezione successiva alla prima. Quindi, nella *target list* si aggiungono tali campi

Nella *range list* si specificano due riferimenti diversi per la tabella LEZIONE poiché sono necessari per indicare due lezioni diverse. Entrambi si riferiscono alla tabella LEZIONE dato che in essa ci sono tutti i dati richiesti.

Nel campo formula ci sono alcune condizioni. Le prime condizioni sono necessarie per indicare che la lezione prima delle ore 17:00 e quella successiva:

- Siano del primo semestre;
- Dell'anno accademico 2016/2017;
- Si svolgono di martedì;
- La prima lezione si svolga prima delle ore 17:00 (è sufficiente l'ora di inizio);
- La successiva (seconda) lezione si svolga dopo l'ora di inizio della prima.

Oltre a queste condizioni, è necessario specificare che la seconda lezione *L2* sia la successiva di *L1*. Per farlo, è possibile utilizzare il quantificatore esistenziale per negare l'esistenza di una lezione tra la prima e la seconda. In questo modo, quest'ultima diventa la successiva. Quindi, le condizioni del semestre, anno accademico, giorno e aula sono le stesse. Al contrario, l'ora di inizio dovrà essere successiva alla prima lezione *L1* e dovrà essere anche precedente alla seconda lezione *L2*. Negando l'esistenza di una lezione con queste condizioni, si ottiene la successione.

7 Tecnologie per le basi di dati

7.1 Transazioni

Una **transazione** identifica un'unità elementare di lavoro svolta da un'applicazione, su cui si vogliono associare particolari proprietà di **correttezza**, **robustezza** e **isolamento**.

Un sistema che mette a disposizione un meccanismo per la definizione e l'esecuzione di transazioni con le caratteristiche suddette viene detto **sistema transazionale**.

La **principale caratteristica** di una transazione è che essa, una volta eseguita, si è certi che non ci saranno esecuzioni parziali. Dunque, l'operazione andrà a buon fine, oppure fallirà senza modificare la base dati.

7.2 Transazioni in SQL

La **sintassi** di una transazione in SQL è la seguente:

```
1 begin transaction
2   <istruzione> | commit work | rollback work
3 end transaction
4
```

Dove al posto di:

- **Istruzione**, possono andare una serie di comandi da eseguire;
- **Commit work** (abbreviazione **commit**), la transazione va a buon fine al raggiungimento di tale operazione;
- **Rollback work** (abbreviazione **rollback**), la transazione non ha alcun effetto al raggiungimento di tale operazione.

Una **transazione** è **ben formattata** se rispetta le seguenti caratteristiche:

1. Inizia con un'istruzione **begin transaction**;
2. Termina con un'istruzione **end transaction**;
3. L'esecuzione incontra un **commit** o un **rollback** e successivamente non esegue altri accessi alla base di dati.

Esempio di transazione ben formata:

```
1 begin transaction;
2   update ContoCorrente
3     set Saldo = Saldo + 10
4     where NumConto = 12202;
5   update ContoCorrente
6     set Saldo = Saldo - 10
7     where NumConto = 42177;
8   select Saldo into A
9     from ContoCorrente
10    where NumConto = 42177;
11  if A >= 0
12    then commit work;
13  else rollback work;
14 end transaction;
15
```

7.3 Proprietà acide delle transazioni

Una transazione possiede quattro **proprietà acide** (Atomicity Consistency Isolation Durability, ACID) che sono:

- **Atomicità** (*Atomicity*)
- **Consistenza** (*Consistency*)
- **Isolamento** (*Isolation*)
- **Persistenza** (*Durability*)

7.3.1 Atomicità

L'**atomicità** rappresenta il fatto che una transazione è un'unità **indivisibile** di esecuzione. Quindi, o vengono resi visibili tutti gli effetti di una transazione, oppure essa non deve aver alcun effetto sulla base di dati (tutto o niente).

Per applicare questa proprietà, si implementano due implicazioni:

- **Transazione interrotta prima** del **commit**, il sistema deve ricostruire la situazione esistente prima dell'esecuzione della transazione eliminando il lavoro eseguito fino a quel momento.
- **Transazione interrotta dopo** il **commit**, il sistema deve assicurare che la transazione lasci la base di dati nel suo stato finale.

7.3.2 Consistenza

La **consistenza** richiede che l'esecuzione della transazione non violi i vincoli di integrità definiti sulla base di dati. Nel caso di una violazione, il sistema interviene per annullare la transazione o per correggere la violazione del vincolo. Il controllo della violazione può essere di due tipi:

- **Controllo della violazione immediata.** I controlli vengono eseguiti durante l'esecuzione della transazione. Così facendo, è possibile rimuovere gli effetti della specifica istruzione di manipolazione dei dati che causa la violazione del vincolo, senza imporre un aborto delle transazioni.
- **Controllo della violazione differita.** I controlli vengono eseguiti al termine dell'esecuzione della transazione, ovvero dopo il *commit*. Nel caso in cui ci sia una violazione, viene abortita l'intera transazione.

7.3.3 Isolamento

L'**isolamento** richiede che l'esecuzione di una transazione sia indipendente dalla contemporanea esecuzione di altre transazioni.

Per applicare questa proprietà, si implementano due implicazioni:

- **Esecuzione concorrente** di un insieme di transazioni deve essere analogo al risultato che le stesse transazioni otterrebbero qualora ciascuna di esse fosse eseguita da sola.
- **Esecuzione indipendente.** L'esecuzione indipendente di ogni transazione evita che un eventuale *rollback* provochi un effetto domino generando una *rollback* anche nelle altre.

7.3.4 Persistenza

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

L'applicazione di questa proprietà garantisce gli effetti delle transazioni che, al momento di un eventuale guasto, abbiano già eseguito un *commit*.

7.4 Architettura di riferimento di un DBMS

Si lascia qui di seguito l'immagine di un'architettura di un DBMS. Non è necessario approfondire più di tanto tale argomento, ma è utile sapere della sua conoscenza. L'unica cosa **importante** da ricordare è **quali sono i moduli che contribuiscono a garantire le proprietà delle transizioni**:

- Gestore dei metodi d'accesso:
 - Consistenza
- Gestore dell'esecuzione concorrente:
 - Atomicità
 - Isolamento
- Gestore dell'affidabilità
 - Atomicità
 - Persistenza

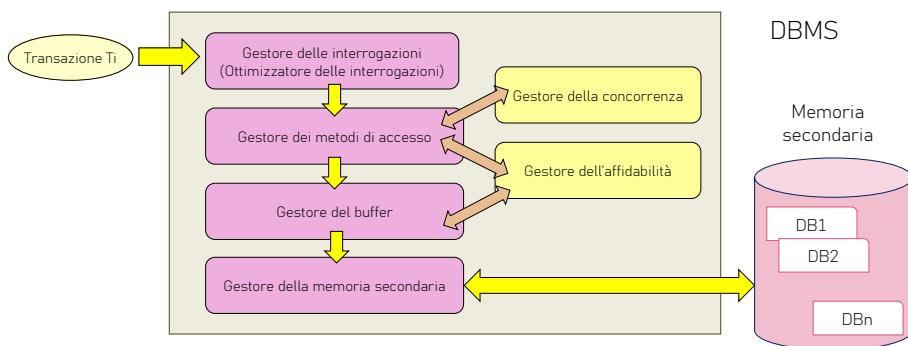


Figura 24: Architettura generale di un DBMS.

7.4.1 Gestore (ottimizzatore) delle interrogazioni

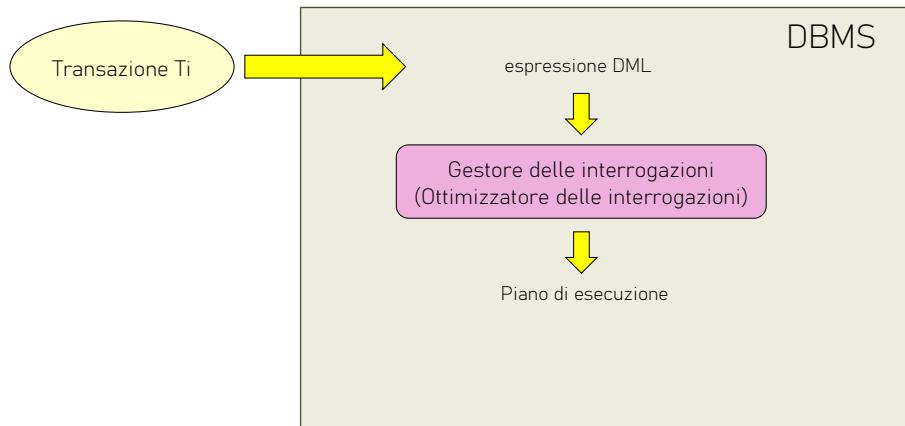


Figura 25: Gestore (ottimizzatore) delle interrogazioni.

7.4.2 Gestore dei metodi d'accesso e dell'esecuzione concorrente

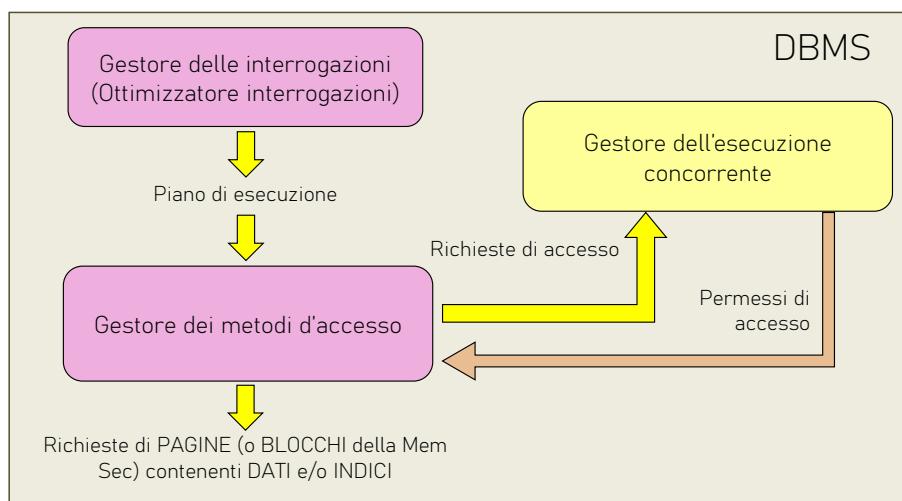


Figura 26: Gestore dei metodi d'accesso e dell'esecuzione concorrente.

7.4.3 Gestore dei metodi d'accesso e dell'affidabilità

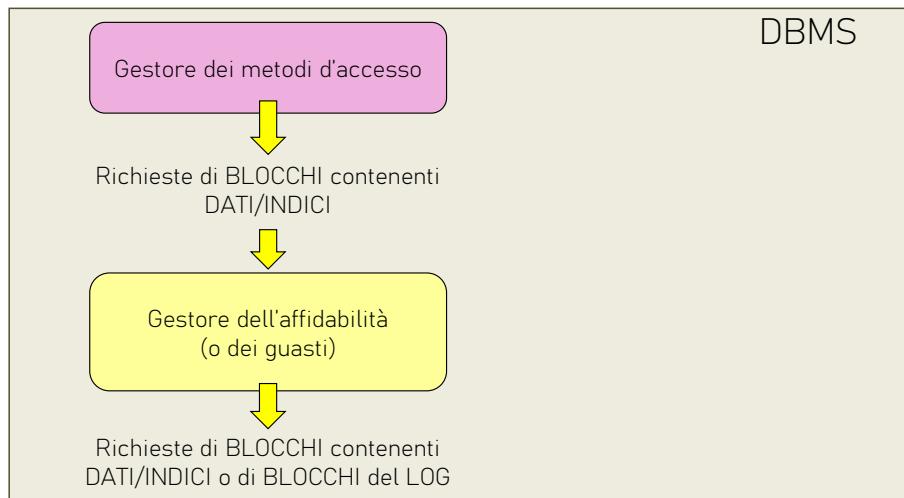


Figura 27: Gestore dei metodi d'accesso e dell'affidabilità.

7.4.4 Gestore dell'affidabilità e del buffer

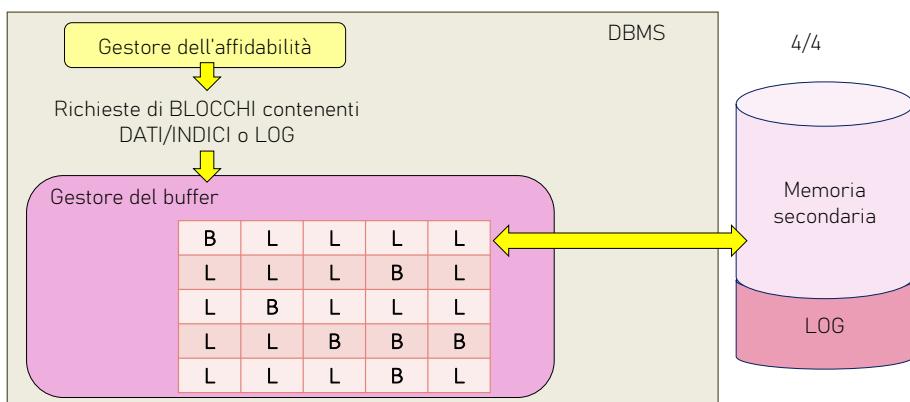


Figura 28: Gestore dell'affidabilità e del buffer.

8 Approfondimento gestore dell'affidabilità e del buffer

8.1 Memoria secondaria

Le basi di dati hanno la necessità di **gestire dati in memoria secondaria per due motivi**:

1. La **dimensione della memoria primaria** (e.g. RAM) non è sufficiente per contenere interamente una base di dati;
2. Le basi di dati hanno come **caratteristica fondamentale la persistenza**. Essa è possibile **applicarla solo su una memoria** di massa (**secondaria**) sulla quale i dati non possono essere persi.

Si elencano alcune **caratteristiche** delle memorie secondarie:

- Una memoria secondaria non può essere utilizzata direttamente dai programmi, ma prima è necessario un passaggio attraverso la memoria primaria;
- I dati sono organizzati in **blocchi** di dimensione fissa o variabile a seconda del sistema;
- Le operazioni ammesse sono due: lettura e scrittura di un blocco;
- Il tempo di lettura/scrittura di un blocco è molto più alto rispetto ad un accesso e ad un'elaborazione dei dati in memoria centrale (e.g. RAM). Quindi, solitamente si conteggiano solo gli accessi alla memoria secondaria, quindi il numero di blocchi letti o scritti.

8.2 Gestione del buffer

L'interazione tra la memoria centrale e la memoria secondaria è realizzata nel DBMS attraverso l'utilizzo di una **zona della memoria centrale** chiamata **buffer** e **condivisa con tutte le applicazioni**.

L'**obiettivo** del *buffer* è quello di evitare di ripetere accessi multipli alla memoria secondaria per tutti quei dati che vengono utilizzati più volte in un tempo ravvicinato. La sua gestione è dunque fondamentale.

Un *buffer* è **organizzato in pagine di dimensione pari a un numero intero di blocchi**. Il **gestore del buffer** si occupa del **caricamento** e dello **scaricamento** (salvataggio) delle pagine dalla memoria centrale alla memoria di massa. Per semplicità, si può dire che ogni caricamento o salvataggio di pagina richiede un'operazione su memoria di massa (lettura o scrittura rispettivamente).

Quindi, le **operazioni** sono:

- **Lettura**, lettura dal buffer se presente, altrimenti lettura fisica;
- **Scrittura**, gestore del buffer differisce la scrittura fisica se tale attesa è compatibile con la proprietà di affidabilità del sistema, ovvero se si è sicuri che l'operazione vada a buon fine.

La gestione del buffer segue il **principio di località dei dati**: i dati referenziati di recente hanno maggior probabilità di essere referenziati nuovamente nel futuro. Ne consegue che i buffer contengono le pagine sulle quali vengono fatte la maggior parte degli accessi. Inoltre, una **legge empirica** afferma che il 20% dei dati è tipicamente acceduto dall'80% delle applicazioni.

Il **gestore del buffer** memorizza alcune **informazioni**:

- Descrizione del contenuto corrente del buffer indicando per *ogni* pagina:
 - Il **file fisico**
 - Il **numero di blocco**
- Mantenimento di alcune variabili di stato:
 - **Contatore** per indicare quanti programmi utilizzano la pagina
 - **Bit di stato** per indicare se la pagina è stata modificata

8.3 Primitive per la gestione del buffer

8.3.1 Primitiva fix

La primitiva **fix** viene utilizzata dalle transazioni per **richiedere l'accesso ad una pagina**. Essa restituisce al chiamante il riferimento (puntatore) alla pagina del buffer, in modo che esso possa accedere effettivamente ai dati. L'esecuzione della primitiva è realizzata nel modo seguente.

1. Viene **cercata la pagina tra quelle in memoria**. Se viene trovata, l'operazione si conclude e l'indirizzo della pagina (puntatore) viene restituito alla transazione richiedente;
2. Se non viene trovata la pagina in memoria, viene **scelta una pagina dal buffer** cercando tra le pagine libere, ovvero con contatore pari a zero. La scelta viene fatta a seconda della strategia adottata, per esempio selezionando la pagina usata meno di recente (LRU, *least recently used*), oppure selezionando quella caricata da più tempo (FIFO, *first input first output*). Inoltre, nel caso in cui il bit di stato segnala che la pagina è stata modificata, essa viene aggiornata in memoria di massa (operazione di *flush*). Infine, viene identificata la pagina da caricare nel buffer e successivamente avviene l'operazione di lettura.
3. Se non esistono pagine libere, il gestore del buffer può scegliere due approcci differenti:
 - (a) Approccio **steal**, viene sottratta una pagina a un'altra transazione. La pagina sottratta viene chiamata vittima e viene scaricata in memoria di massa (operazione di *flush*). Infine, vengono eseguite le operazioni di conversione di indirizzi e successivamente l'operazione di lettura.
 - (b) Approccio **non steal**, la transazione viene sospesa, in attesa che si liberino pagine dal buffer. Quindi, essa entra in una coda di transazioni, la quale è mantenuta dal gestore del buffer. Nel momento in cui si libera una pagina, viene identificata, caricata nel buffer e successivamente avviene l'operazione di lettura.

In ogni caso, **quando si effettua un accesso ad una pagina, viene incrementato il contatore relativo all'utilizzo della pagina**.

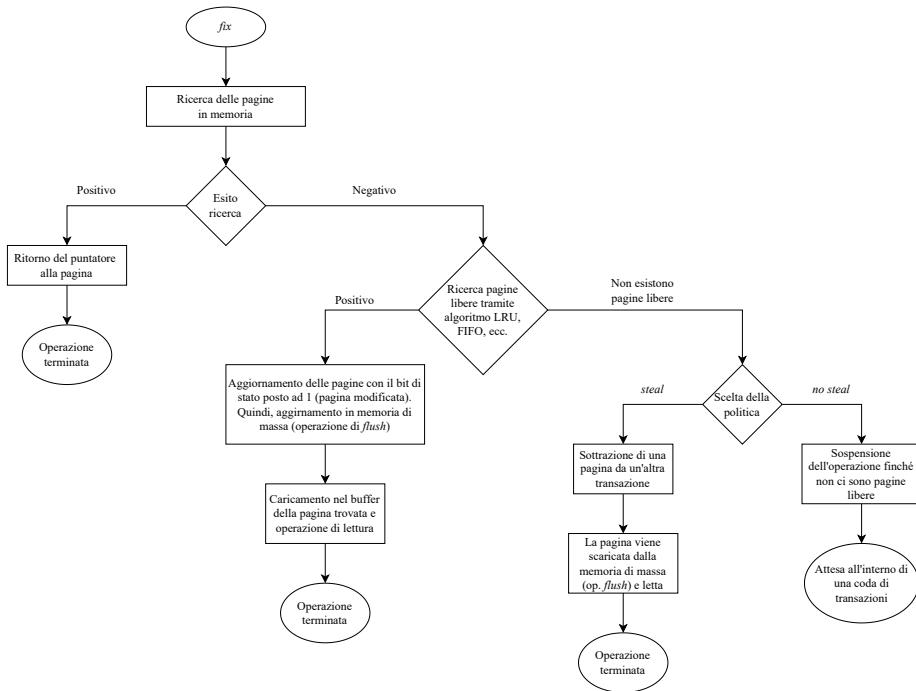


Figura 29: Diagramma a blocchi della primitiva fix

8.3.2 Primitiva `setDirty`

La primitiva `setDirty` indica al gestore del buffer che **una pagina è stata modificata**. Il suo effetto è la modifica del bit di stato relativo.

8.3.3 Primitiva `unfix`

La primitiva `unfix` indica al gestore del buffer che il modulo **chiamante ha terminato di usare la pagina**. Il suo effetto è decrementare il contatore di utilizzo della pagina.

8.3.4 Primitiva `force`

La primitiva `force` trascrive in memoria di massa, in modo sincrono, una **pagina del buffer**.

8.4 Gestore dell'affidabilità

Il **controllo dell'affidabilità** garantisce due **proprietà fondamentali** delle transazioni: atomicità e persistenza.

- **Atomicità**, garantire che le transazioni non vengano lasciate incomplete
- **Persistenza**, garantire che gli effetti di ciascuna transazione conclusa con un *commit* siano mantenuti in modo permanente

Il gestore dell'affidabilità garantisce queste proprietà tramite il **log**, ovvero un archivio persistente su cui registra le varie azioni svolte dal DBMS.

L'**architettura** del controllore di affidabilità è descritta nella seguente immagine.

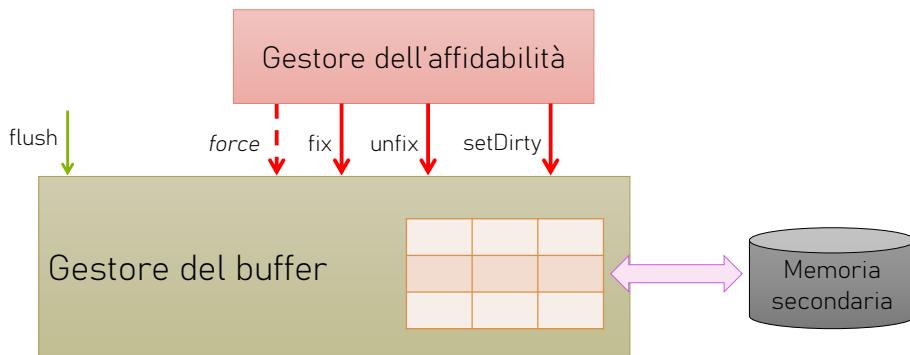


Figura 30: Architettura di un gestore dell'affidabilità.

Il controllore dell'affidabilità riceve richieste di accessi a pagine in lettura e scrittura, che passa al *buffer manager*, e genera altre richieste di lettura e scrittura di pagine necessarie a garantire la robustezza e la resistenza ai guasti.

Inoltre, questa componente per eseguire il suo compito necessita di una **memoria stabile**, ovvero una memoria che risulti resistente ai guasti. Ovviamente, non esiste un dispositivo con tali caratteristiche, ma grazie ad alcuni meccanismi, è possibile rendere tale probabilità prossima allo zero.

8.4.1 Definizione log

Il **log** è un file sequenziale di cui è responsabile il controllore dell'affidabilità, scritto in memoria stabile e contenente informazione ridondante che **consente di ricostruire il contenuto della base di dati a seguito di guasti**. L'ordine di salvataggio delle varie transazioni è quello **temporale**.

Ogni salvataggio produce un *record* nel file, il quale può essere di due **tipi**:

1. **Record di transazione**, identificano le **attività svolte da ciascuna transazione**. Quindi, ogni transazione inserisce nel log un record di **begin** (corrispondente al *start transaction*), vari record relativi alle azioni effettuate (corrispondente ad *insert*, *delete*, *update*) e un record di **commit** oppure di **abort** (corrispondente a *rollback*).
2. **Record di sistema**, indicano l'**effettuazione di operazioni** specifiche del controllore dell'affidabilità che in totale sono due: ***dump*** e ***checkpoint***.

8.4.2 Notazione dei record nel log

I **record che descrivono una transizione** vengono identificati nel seguente modo:

- I record:

- **begin** $\longrightarrow B(T)$
- **commit** $\longrightarrow C(T)$
- **abort** $\longrightarrow A(T)$

Contengono l'indicazione del tipo di record e l'identificativo T della transazione.

- Il record:

- **update** $\longrightarrow U(T, O, BS, AS)$

Contiene l'identificativo T della transazione, l'identificativo O dell'oggetto su cui avviene l'*update*, e poi due valori BS e AS che descrivono rispettivamente il valore dell'oggetto O precedentemente alla modifica (*before state*), e successivamente alla modifica (*after state*).

- I record:

- **insert** $\longrightarrow I(T, O, AS)$
- **delete** $\longrightarrow D(T, O, BS)$

Sono identici ad **update** ma manca un operatore.

Invece, i **record che consentono di disfare e rifare le rispettive azioni sulla base di dati** sono le seguenti:

- Il record:

- **undo** $\longrightarrow Undo(A)$

Consente di disfare un oggetto O ricopiando in O il valore BS (*before state*); l'**insert** viene disfatto cancellando l'oggetto O .

- Il record:

- **redo** $\longrightarrow Redo(A)$

Consente di rifare un'azione su un soggetto O ricopiando in O il valore AS (*after state*); il **delete** viene rifatto cancellando l'oggetto O .

Dato che le operazioni ***Undo*** e ***Redo*** sono definite tramite un'azione di copiatura, vale una proprietà essenziale: la **proprietà di idempotenza**. Essa afferma che l'effettuazione di un numero arbitrario di undo o redo della stessa azione equivale allo svolgimento di tale azione una sola volta:

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

8.4.3 Checkpoint e dump

Nonostante i record introdotti nel paragrafo precedente siano sufficienti a svolgere un'azione di ripristino, si introducono due nuovi operatori per consentire un ripristino molto più rapido.

Un **checkpoint** è un'operazione svolta periodicamente dal gestore dell'affidabilità, con l'obiettivo di registrare quali transazioni sono attive.

La sua **notazione** è $CK(T_1, T_2, \dots, T_n)$ dove T_1, T_2, \dots, T_n denotano gli identificatori delle transazioni attive all'istante del checkpoint.

L'operazione si articola in quattro passaggi:

1. Viene **sospesa l'accettazione di operazioni** di scrittura, commit o abort, da parte di ogni transazione.
2. Viene **trasferita** in memoria di massa (operazione di *force*) tutte le **pagine del buffer su cui sono state eseguite modifiche** da parte di transazioni che hanno già effettuato il **commit**.
3. **Scrittura** in modo sincrono (*force*) **nel log un record di checkpoint** contenente gli identificatori delle transazioni attive.
4. **Ripresa dell'accettazione** delle operazioni sopra sospese.

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. Al termine di tale operazione, viene scritto nel log un **record di dump**, il quale segnala la presenza di una copia eseguita in un determinato istante.

La sua **notazione** è semplicemente *DUMP*.

8.4.4 Esecuzione delle transazioni e scrittura dei log

Il gestore dell'affidabilità deve garantire che siano seguite due regole durante il funzionamento delle transazioni. Esse **definiscono i requisiti minimi che consentono di ripristinare la correttezza della base di dati a fronte di guasti.**

- **Regola WAL** (*Write Ahead Log*) impone che la parte *before state* (pre modifica) dei record di un log venga **scritta nel log prima di effettuare la corrispondente operazione sulla base di dati**.

Questa regola **consente** di disfare (*undo*) le scritture già effettuate in memoria di massa da parte di una transazione che non ha ancora effettuato un commit, poiché per ogni aggiornamento viene reso disponibile in modo affidabile il valore precedente la scrittura.

- **Regola di Commit-Precedenza** impone che la parte *after state* (post modifica) dei record di un log venga **scritta nel log prima di effettuare il commit**.

Questa regola **consente** di rifare (*redo*) le scritture già decise da una transazione che ha effettuato il commit ma le cui pagine modificate non sono ancora state trascritte dal buffer manager in memoria di massa.

Se il guasto avviene:

- **Prima della scrittura nel log del record di commit**, allora un eventuale guasto comporta l'*undo* delle azioni effettuate, ricostruendo lo stato iniziale della base di dati.
- **Dopo la scrittura nel log del record di commit**, allora un eventuale guasto comporta il *redo* delle azioni effettuate, in modo da da ricostruire con certezza lo stato finale della transazione.

8.4.5 Gestione dei guasti

I **guasti** si suddividono in **due classi**:

- **Guasti di sistema** sono guasti indotti da “bachi software”, per esempio del sistema operativo.
Perdita di: contenuto della memoria centrale (e dunque di tutti i buffer);
Sopravvivenza di: rimane valido il contenuto della memoria di massa (e quindi della base di dati e del log).
- **Guasti di dispositivo** sono guasti relativi ai dispositivi di gestione della memoria di massa, per esempio lo strisciamento delle testine di un disco.
Perdita di: contenuto della base di dati;
Sopravvivenza di: rimane valido il log.

Il log è l'unico file che sopravvive ai guasti. Per cui è importante mantenere la sua integrità e non perderlo.

La **gestione ideale dei guasti** è chiamata ***fail-stop***: nel momento in cui il sistema individua un guasto (di sistema o di dispositivo), viene forzato immediatamente un arresto completo della transazioni. Dopodiché, viene effettuato un ripristino del corretto funzionamento del sistema operativo (*boot*). Quest'ultima operazione comporta un buffer completamente vuoto e un sistema riutilizzabile.

Le procedure adottate a seconda del guasto sono due: **riprresa a caldo** (*warm restart*) nel caso di guasto di sistema, **riprresa a freddo** (*cold restart*) nel caso di guasto di dispositivo.

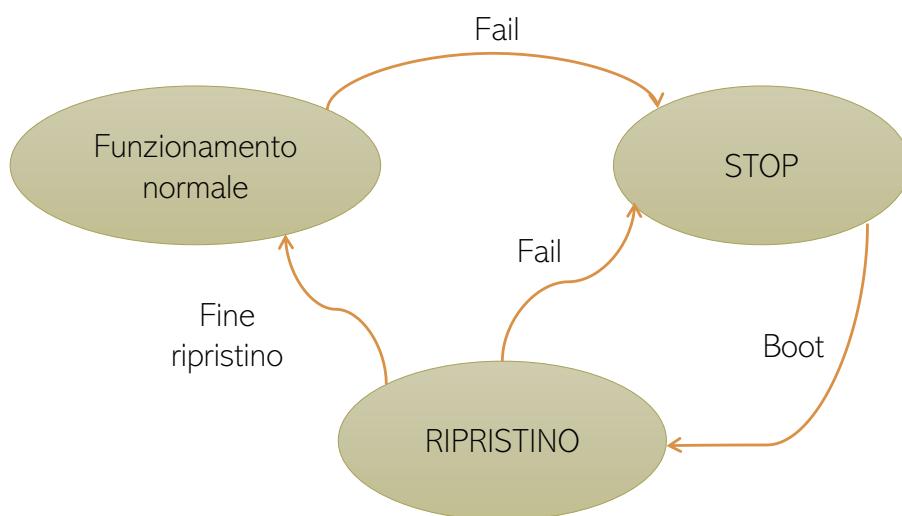


Figura 31: Modello *fail-stop* di funzionamento di un DBMS.

8.4.6 Ripresa a caldo

La **ripresa a caldo** è articolata in quattro fasi:

1. Viene eseguito l'accesso all'ultimo blocco del log, ovvero quello corrente all'istante del guasto. Dopodiché viene ripercorso all'indietro il log fino all'ultimo (più recente) record di checkpoint.
2. Vengono decise le transazioni da rifare (*redo*) o disfare (*undo*).
Vengono costruiti due insiemi UNDO e REDO contenenti identificativi di transazioni:
 - UNDO inizializzato con le transazioni attive al checkpoint.
 - REDO inizializzato con l'insieme vuoto.

Viene ripercorso il file log in avanti (dopo essere tornati indietro al punto uno), aggiungendo:

- Ad UNDO tutte le transazioni di cui è presente un record di *begin*;
- A REDO tutti gli identificativi delle transazioni, quindi spostandoli da UNDO a REDO, di cui è presente il record di *commit*.

Al termine il risultato sarà:

- UNDO contenente tutti gli identificativi delle transazioni da disfare
 - REDO contenente tutti gli identificativi delle transazioni da rifare.
3. Viene ripercorso nuovamente all'indietro il file log disfacendo le transazioni dall'insieme UNDO, risalendo fino alla prima azione della transazione più “vecchia” nei due insiemi UNDO e REDO. Questa azione potrebbe precedere il record di checkpoint nel log.
 4. Infine, viene ripercorso in avanti fino alla fine, così da applicare le azioni di *redo* nell'ordine in cui sono state registrate nel log. In questo modo viene replicato esattamente il comportamento delle transazioni originali.

Questo processo può essere visto come: ripercorrere il file fino all'ultimo backup eseguito per inserire un break-point; creare due insiemi, ripercorrendo il file fino alla fine, contenenti le azioni da rieseguire e le azioni già confermate e dunque da eliminare; ripercorrere nuovamente il file fino al break-point eliminando le varie operazioni eseguite e confermate prima del guasto; ripercorrere per un'ultima volta il file fino alla fine rieseguendo le operazioni non confermate grazie all'insieme creato.

È consigliato verificare l'algoritmo con l'esempio a fine capitolo per comprendere meglio il suo funzionamento.

8.4.7 Ripresa a freddo

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

1. Accesso al file *dump* e viene ricopiato selettivamente la parte deteriorata della base di dati.
2. Viene ripercorso il file di log fino al record di *dump* e successivamente viene ripercorso fino alla fine applicando relativamente alla parte deteriorata sia le azioni sulla base di dati, sia le azioni di commit o abort. Viene così ripristinata la situazione prima del guasto.
3. Infine, viene svolta una ripresa a caldo per confermare le operazioni interrotte.

8.4.8 Esercizio sulla ripresa a caldo

Si supponga che nel log vengano registrate le seguenti azioni:

$$\begin{aligned} & B(T1), B(T2), U(T2, O1, B1, A1), I(T1, O2, A2), B(T3), C(T1), B(T4), \\ & U(T3, O2, B3, A3), U(T4, O3, B4, A4), CK(T2, T3, T4), C(T4), B(T5), \\ & U(T3, O3, B5, A5), U(T5, O4, B6, A6), D(T3, O5, B7), A(T3), C(T5), \\ & \quad I(T2, O6, A8) \end{aligned}$$

Si verifica un guasto. Il protocollo di ripresa a caldo esegue le seguenti operazioni:

- I. Viene risalito il log fino all'ultimo checkpoint, ovvero il più recente:

$$\begin{aligned} & \textcolor{red}{CK(T2, T3, T4)}, C(T4), B(T5), U(T3, O3, B5, A5), U(T5, O4, B6, A6), \\ & \quad D(T3, O5, B7), A(T3), C(T5), I(T2, O6, A8) \end{aligned}$$

Vengono inizializzati i due insiemi:

$$UNDO = \{T2, T3, T4\} \quad REDO = \{\}$$

- II. Viene ripercorso il file log in avanti aggiornando man mano i due insiemi. In *REDO* vengono inseriti tutti gli identificativi che hanno eseguito il *commit* e in *UNDO* tutti gli identificativi che iniziano (*begin*) una transazione:

- (a) Incontro con *C(T4)*, aggiornamento insiemi:

$$UNDO = \{T2, T3\} \quad REDO = \{T4\}$$

- (b) Incontro con *B(T5)*, aggiornamento insiemi:

$$UNDO = \{T2, T3, T5\} \quad REDO = \{T4\}$$

- (c) Incontro con *C(T5)*, aggiornamento insiemi:

$$UNDO = \{T2, T3\} \quad REDO = \{T4, T5\}$$

- III. Viene ripercorso nuovamente indietro il file log fino all'azione della transazione più vecchia nei due insiemi. Le azioni che riguardano gli identificativi dell'insieme *UNDO* sono:

$$\begin{aligned} & B(T1), B(T2), \textcolor{red}{U(T2, O1, B1, A1)}, I(T1, O2, A2), B(T3), C(T1), B(T4), \\ & \textcolor{red}{U(T3, O2, B3, A3)}, U(T4, O3, B4, A4), CK(T2, T3, T4), C(T4), B(T5), \\ & \textcolor{red}{U(T3, O3, B5, A5)}, U(T5, O4, B6, A6), \textcolor{red}{D(T3, O5, B7)}, A(T3), C(T5), \\ & \quad \textcolor{red}{I(T2, O6, A8)} \end{aligned}$$

Dato che le operazioni sono in ordine temporale, la più vecchia è la *U(T2, O1, B1, A1)*. Quindi, risalendo fino a tale operazione, vengono disfatte nel frattempo le transazioni dall'insieme *UNDO* eseguendo le operazioni al contrario così da riprendere lo stato prima delle modifiche:

- (a) Incontro con $I(T2, O6, A8)$, ripristino eseguendo operazione di eliminazione:

$$Delete(O6)$$

- (b) Incontro con $D(T3, O5, B7)$, ripristino eseguendo operazione di inserimento:

$$Insert(O5) \quad \text{con } O5 = B7$$

- (c) Incontro con $U(T3, O3, B5, A5)$, ripristino eseguendo operazione di assegnamento del valore *before state*:

$$O3 = B5$$

- (d) Incontro con $U(T3, O2, B3, A3)$, ripristino eseguendo operazione di assegnamento del valore *before state*:

$$O2 = B3$$

- (e) Incontro con $U(T2, O1, B1, A1)$, ripristino eseguendo operazione di assegnamento del valore *before state*:

$$O1 = B1$$

IV. Infine, viene ripercorso il file fino alla fine svolgendo le azioni di *REDO*. Le azioni che riguardano gli identificativi dell'insieme *REDO* sono:

$$\begin{aligned} & B(T1), B(T2), U(T2, O1, B1, A1), I(T1, O2, A2), B(T3), C(T1), B(T4), \\ & U(T3, O2, B3, A3), \textcolor{red}{U(T4, O3, B4, A4)}, CK(T2, T3, T4), C(T4), B(T5), \\ & U(T3, O3, B5, A5), \textcolor{red}{U(T5, O4, B6, A6)}, D(T3, O5, B7), A(T3), C(T5), \\ & I(T2, O6, A8) \end{aligned}$$

Partendo dall'operazione $U(T2, O1, B1, A1)$, si eseguono le operazioni di *REDO* ripristinando gli stati in ordine temporale:

- (a) Incontro con $U(T4, O3, B4, A4)$, esecuzione dell'operazione di assegnamento del valore *after state*:

$$O3 = A4$$

- (b) Incontro con $U(T5, O4, B6, A6)$, esecuzione dell'operazione di assegnamento del valore *after state*:

$$O4 = A6$$

9 Gestore dei metodi d'accesso

Il **gestore dei metodi d'accesso** è colui che esegue il piano di esecuzione prodotto dall'ottimizzatore e produce sequenze di accessi ai blocchi della base di dati presenti in memoria secondaria. Con **metodi di accesso** si intende tutti i moduli software che implementano gli algoritmi di accesso e manipolazione dei dati organizzati in specifiche strutture fisiche.

9.1 Gestione delle tuple nelle pagine

Ogni organizzazione fisica è differente a seconda dell'architettura, tuttavia sotto molti aspetti hanno caratteristiche in comune.

In ogni pagina sono presenti: l'**informazione utile**, cioè i dati veri e propri, e l'**informazione di controllo**, quella che consente di accedere all'informazione utile. Inoltre, sono presenti anche informazioni:

- Ogni pagina, corrispondente ad un blocco di memoria di massa, ha due parti contenenti l'**informazione di controllo utilizzata dal file system**:
 - Una **parte iniziale** (*block header*)
 - Una **parte finale** (*block trailer*)
- Ogni pagina, contenente dati gestiti dal DBMS, ha altre due parti contenenti l'**informazione di controllo relativa alla specifica struttura fisica**³:
 - Una **parte iniziale** (*page header*)
 - Una **parte finale** (*page trailer*)
- Ogni pagina ha il suo **dizionario di pagina**, il quale contiene:
 - **Puntatori** a ciascun dato utile elementare contenuto nella pagina;
 - **Parte utile**, la quale contiene i dati.

Solitamente, queste due parti crescono come uno stack (si veda l'immagine) contrapposti, lasciando memoria libera in uno spazio contiguo compreso tra i due stack.

- Ogni pagina contiene **bit di parità** per verificare che l'informazione in essa contenuta sia valida.

³Questa informazione può per esempio contenere l'identificatore dell'oggetto (tabella, indice, dizionario dei dati, ecc.) contenuto nella pagina, puntatori a pagine successive o precedenti nella struttura dati, numero di dati utili elementari (tuple) contenuti nella pagina, quantità di memoria libera disponibile nella pagina

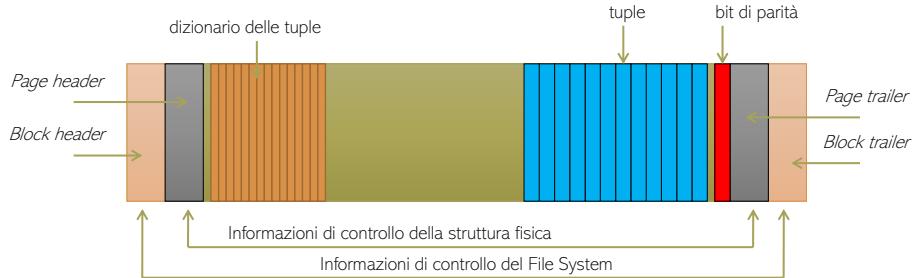


Figura 32: Organizzazione di una pagina.

In riferimento al **dizionario di pagina**, esso ha una struttura variabile a seconda della struttura delle tuple:

- **Tuple con lunghezza fissa**, la struttura dei dizionari è semplificata, ma si corre il rischio di spreco di spazio nelle pagine.
- **Tuple con lunghezza variabile**, il dizionario di pagina contiene l'indicazione degli *offset* di ciascuna tupla rispetto all'inizio della parte utile e di ciascun valore dei vari campi presenti nella tupla rispetto all'inizio della tupla stessa.

Le **operazioni** (primitive) eseguibili sulle pagine sono:

- **Inserimento e aggiornamento** che vengono eseguite in modo differente a seconda dello spazio disponibile:
 - Spazio non contiguo, le operazioni sono precedute da una riorganizzazione della pagina con conseguente accesso in memoria centrale;
 - Spazio contiguo, le operazioni vengono eseguite senza difficoltà;
 - Spazio insufficiente, le operazioni comportano un'interazione con il file system, che deve allocare nuovi blocchi per il file.
- **Cancellazione**, operazione sempre eseguibile.
- **Accesso a una particolare multipla**, identificata tramite il valore della chiave oppure in base al suo *offset*.
- **Accesso a un campo di una particolare tupla**, identificata la tupla come nel punto precedente (tramite chiave o *offset*) e successivamente viene identificato il campo in base all'*offset* e alla lunghezza del campo stesso.

9.2 Strutture primarie per l'organizzazione di file

Le principali tecniche utilizzate per la struttura primaria di un file, cioè quella che stabilisce il criterio secondo il quale sono disposte le tuple nell'ambito del file. Le strutture possono essere divise in tre categorie principali:

1. **Sequenziali.** Un file è costituito da vari blocchi di memoria “logicamente” consecutivi, e le tuple vengono inserite nei blocchi rispettando una sequenza:

- Organizzazione **disordinata (seriale)**, la sequenza delle tuple è indotta dal loro ordine di immissione;
- Organizzazione **ad array**, le tuple sono disposte come in un array e la loro posizione dipende dal valore assunto in ciascuna tupla da un campo di **indice**;
- Organizzazione **sequenziale ordinata**, la sequenza delle tuple dipende dal valore assunto in ciascuna tupla da un campo del file.

2. **Ad accesso calcolato (hash)**

3. **Ad albero**
-

9.3 Strutture sequenziali

9.3.1 Struttura sequenziale ordinata

La **struttura sequenziale ordinata** prevede la memorizzazione dei record secondo un ordinamento fisico coerente con l'ordinamento di un campo chiamato **chiave**⁴.

Le strutture ordinate rendono **efficienti** le operazioni che hanno bisogno proprio dell'ordinamento utilizzato, come per esempio la produzione di un elenco ordinato per cognome. Inoltre, esse **favoriscono** le “selezioni su intervallo” (**range query**), per esempio la **ricerca di tuple** con un cognome che inizia con una certa sequenza di lettere oppure con una data compresa in un certo intervallo. Questo accade poiché memorizzano in posizioni consecutive i record che soddisfano la condizione: una volta individuato il primo record (per esempio tramite un indice), l'accesso agli altri sarà molto efficiente.

Le strutture ordinate presentano anche **inconvenienti** in presenza di:

- **Aggiornamenti**, a causa della necessità di mantenere l'ordinamento;
- **Eliminazioni**, provocano spreco di spazio;
- **Inserimenti in posizioni intermedie**, richiedono spazio aggiuntivo con perdita di contiguità.

Per questi motivi, le strutture possono richiedere **periodicamente riorganizzazioni**.

⁴In questo caso ci si riferisce ad un campo di ordinamento, ma potrebbero essere due o più, con ordinamento sulla base del primo e, in caso di valori uguali, sul secondo, e così via. Inoltre, il campo su cui è realizzato l'ordinamento non è necessariamente la chiave della relazione.

9.3.2 Indici primari e secondari

L'**obiettivo** degli indici è quello di consentire ricerche efficienti.

Dato un file f con un campo chiave k , un **indice secondario** è un altro file, in cui ciascun record è logicamente composto da due campi:

- Uno contenente un valore della chiave k del file f ;
- Uno contenente l'indirizzo o gli indirizzi fisici dei record di f che hanno quel valore di chiave.

L'indice secondario è **ordinato in base al valore della chiave** e quindi consente una rapida ricerca in base a tale valore. Per cui, l'indice secondario può essere usato da un programma per accedere rapidamente ai dati del file primario.

Inoltre, un indice secondario **deve** contenere i riferimenti a tutti i valori della chiave, visto che record con valori consecutivi della chiave possono trovarsi in blocchi ben diversi. Per questo motivo viene detto che è un **indice denso**.

Se l'indice contiene al suo interno i dati oppure è realizzato su un file ordinato sullo stesso campo su cui è definito l'indice stesso, allora viene definito **indice primario**. Questo perché viene garantito un accesso in base alla chiave e sono contenuti anche i record fisici necessari per memorizzare i dati o comunque ne vincola l'allocazione. Quindi, un **file non può avere un solo indice primario e sequenziale**.

Per l'indice primario, è possibile utilizzare:

- L'**indice denso**, per ogni occorrenza della chiave presente nel file esiste un record corrispondente nell'indice;
- L'**indice sparso**, solo per alcune occorrenze della chiave presenti nel file esiste un corrispondente record nell'indice, tipicamente uno per blocco.

9.3.3 Esempi di indice primario e secondario

Qui di seguito si lascia lo schema di un esempio di applicazione di indice primario. Il file sequenziale è diviso in due blocchi e ciascuno è puntato da un indice denso e/o sparso.

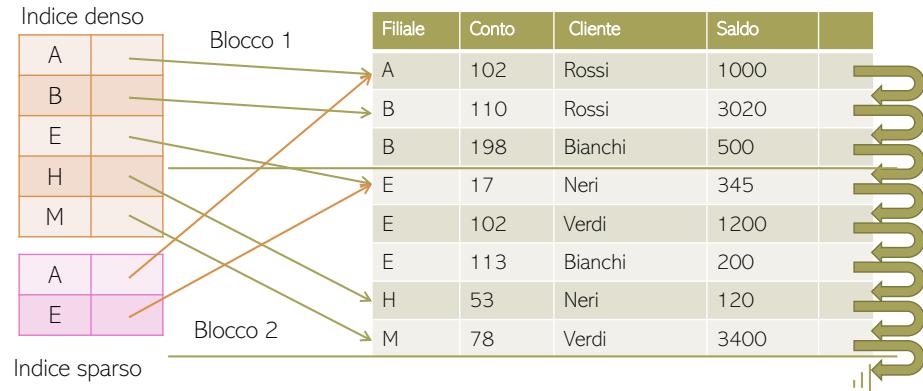


Figura 33: Esempio di indice primario.

L’indice primario in versione “sparso”, punta al primo elemento di ogni blocco, in questo modo è possibile risalire agli altri grazie alla concatenazione delle tuple nei blocchi.

Invece, la versione “densa”, punta al primo elemento di ogni riferimento e non ad un solo riferimento di un blocco.

Al contrario, nel seguente esempio è possibile osservare l’applicazione di un indice secondario. In questo caso l’indice è solo di tipo “denso”, quindi punta ad ogni riferimento.

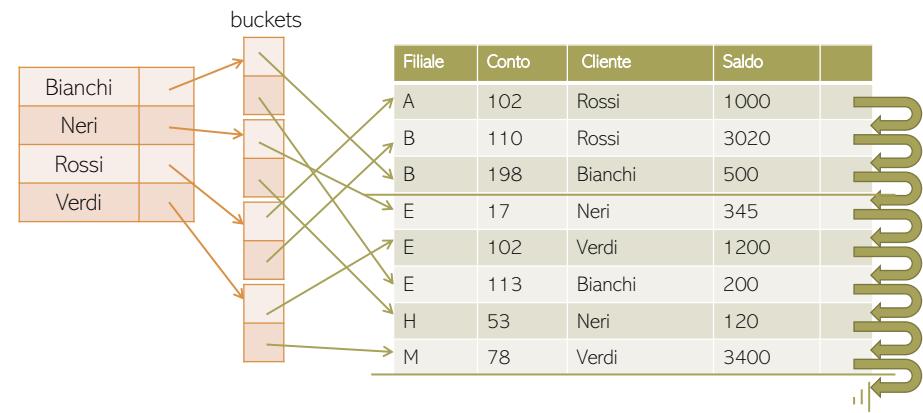


Figura 34: Esempio di indice secondario.

10 Strutture ad albero e hash

10.1 Alberi B+ (*B+-Tree*)

10.1.1 Definizione

Un albero B+ o un ***B+-Tree*** è una tipica struttura ad albero utilizzata nel mondo dell'informatica per rappresentare dei dati. Ogni **nodo** corrisponde ad una **pagina della memoria secondaria**, mentre i **legami** tra nodi diventano **puntatori a pagina**. Ogni **nodo** ha un numero elevato di foglie (figli), quindi generalmente gli alberi hanno pochi livelli, ma molti nodi foglia.

Un albero viene definito **bilanciato** se la lunghezza dei percorsi che collegano la radice ai nodi foglia è costante. Inoltre, viene definito **efficiente** poiché gli inserimenti e le cancellazioni non alterano le prestazioni dell'accesso ai dati.

10.1.2 Struttura di un nodo foglia

La struttura di un **nodo foglia** può essere immaginata come un array. Esso può contenere fino a $n - 1$ valori **ordinati** di chiave di ricerca e fino a n puntatori (quindi il suo **fan-out** è uguale a n). Esiste anche una variante in cui al posto dei valori chiave, il nodo foglia contiene direttamente le tuple (variante implementata in una struttura fisica integrata dati/indice).

p_1	K_1	p_2	...	p_{m-1}	K_{m-1}	p_m
-------	-------	-------	-----	-----------	-----------	-------

Figura 35: Struttura di un nodo foglia.

Nella figura viene evidenziata la sua struttura, dove:

- Le p rappresentano i **puntatori**;
- Le K rappresentano le **chiavi di ricerca** ordinate, quindi dati due indici i, j tale che $i < j$, allora la chiave K_i sarà prima della chiave K_j , ovvero $K_i < K_j$;
- Vale la **relazione** $m \leq n$, dove n è il numero totale di puntatori. Dallo schema è evidente che le chiavi di ricerca possono arrivare al massimo a $m - 1$ rispettando sempre questa condizione;
- Ogni **puntatore** (escluso l'ultimo) **punta alla sua rispettiva chiave**, quindi entrambi avranno gli stessi indici (indice primario);
Ogni **puntatore punta al bucket di puntatori** verso le tuple con chiave K_1 (indice secondario).

L'ultimo puntatore punta al nodo successivo se esiste.

10.1.3 Struttura di un nodo intermedio

Un **nodo intermedio** ha la stessa identica struttura di un nodo foglia, con la differenza che ogni puntatore (in riferimento alla struttura sotto):

- p_1 punta al sottoalbero con chiavi k tali che: $k < K_1$
- p_i punta al sottoalbero con chiavi k tali che: $K_{i-1} < k < K_i$
- p_m punta al sottoalbero con chiavi k tali che: $K_{m-1} \leq k$

p_1	K_1	p_2	\dots	K_{i-1}	p_i	K_i	\dots	p_{m-1}	K_{m-1}	p_m
-------	-------	-------	---------	-----------	-------	-------	---------	-----------	-----------	-------

Figura 36: Struttura di un nodo intermedio.

10.1.4 Vincoli di riempimento

I **vincoli di riempimento** determinano il numero di chiavi e puntatori che sono presenti rispettivamente all'interno di un nodo foglia o di un nodo intermedio.

Un **nodo foglia** che utilizza un vincolo di riempimento con *fan-out* uguale a n , è un nodo contenente un **numero di valori chiave** (rappresentati con il termine $\#chiavi$) vincolato nel seguente modo:

$$\left\lceil \frac{(n-1)}{2} \right\rceil \leq \#chiavi \leq (n-1)$$

Invece, un **nodo intermedio** che utilizza un vincolo di riempimento con *fan-out* uguale a n , è un nodo contenente un **numero di valori puntatori** (rappresentati con il termine $\#puntatori$) vincolato nel seguente modo:

$$\left\lceil \frac{n}{2} \right\rceil \leq \#chiavi \leq n$$

Le parentesi quadre indicano l'arrotondamento all'intero superiore più vicino.

10.1.5 Esempio di albero B+

Il primo caso che viene presentato è il **riempimento minimo dei nodi foglia**.

Il *fan-out* impostato è 4 e i valori delle chiavi presenti sono: $A, B, D, E, F, G, L, M, N, P$.
I vincoli di riempimento sono:

- $2 \leq \#\text{chiavi} \leq 3$
- $2 \leq \#\text{puntatori} \leq 4$

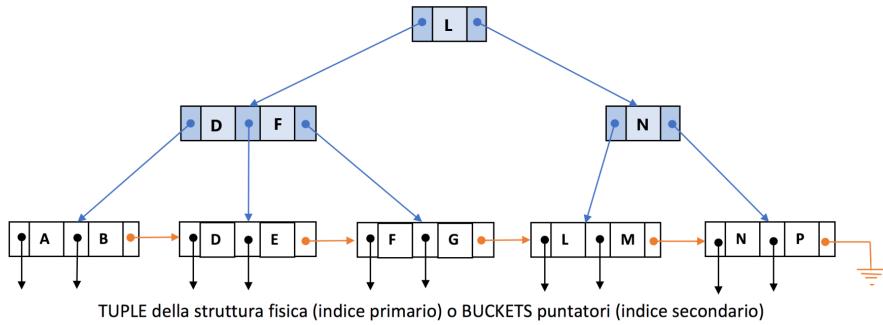
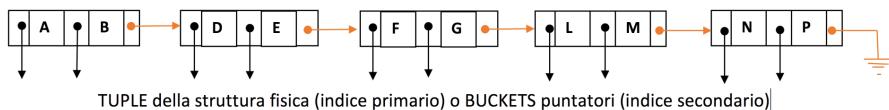
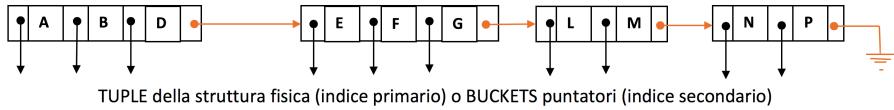


Figura 37: Albero utilizzando un riempimento minimo dei nodi foglia.

Invece, nel secondo caso viene presentato il **riempimento massimo dei nodi foglia**. Il *fan-out* impostato è 4 e i valori delle chiavi presenti sono: $A, B, D, E, F, G, L, M, N, P$. I vincoli di riempimento sono:

- $2 \leq \#\text{chiavi} \leq 3$
- $2 \leq \#\text{puntatori} \leq 4$



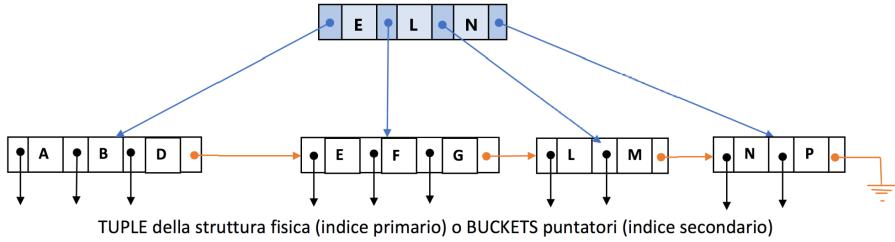


Figura 38: Albero utilizzando un riempimento massimo dei nodi foglia.

10.2 Operazione sugli alberi

10.2.1 Ricerca con chiave K

La **ricerca** tramite una chiave si divide in due passaggi:

1. Ricercare nel **nodo radice** il più piccolo valore di chiave maggiore di K:

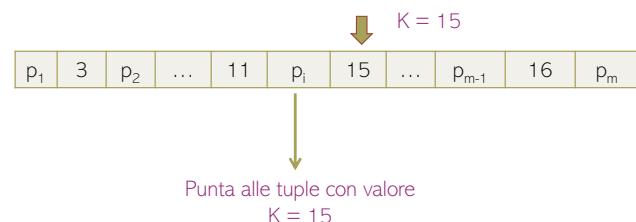
- Se il valore esiste, per esempio K_i , allora viene seguito il puntatore p_i .



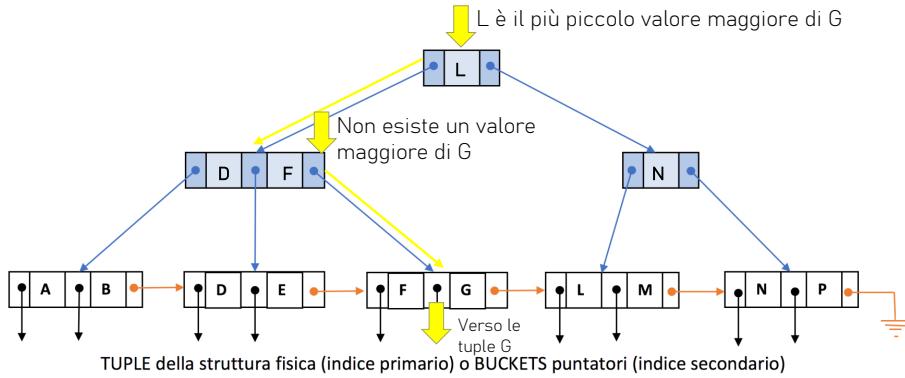
- Se il valore non esiste, allora viene seguito il puntatore p_m .



2. Una volta raggiunto il **nodo foglia**, viene cercato il valore K nel nodo e viene seguito il corrispondente puntatore verso le tuple.



Un **esempio di ricerca** nell'albero B+ è il seguente. In questo caso vengono richieste quelle tuple con valore G della chiave.



Il **costo di una ricerca** nell'indice, in termini di **numero di accessi alla memoria secondaria**, è pari al numero di nodi acceduti nella ricerca.

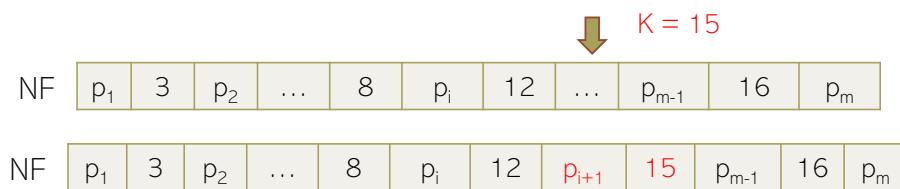
Tale numero, in una struttura ad albero, è **pari alla profondità** ($prof$) dell'albero. Nel caso dei B+ è indipendente dal percorso ed è funzione del *fan-out* n e del numero di valori chiave presenti nell'albero $\#valoriChiave$:

$$prof_{B+-tree} \leq 1 + \log_{\lceil \frac{n}{2} \rceil} \left(\frac{\#valoriChiave}{\lceil \frac{(n-1)}{2} \rceil} \right)$$

10.2.2 Inserimento con chiave K

Come per la ricerca, anche l'**inserimento** tramite una chiave viene eseguito in due passaggi:

1. Ricercare nel nodo foglia (NF) dove il valore K va inserito;
2. Verificare se K è presente o meno nel nodo foglia:
 - Se $K \in NF$:
 - Indice primario: non compie nessuna azione;
 - Indice secondario: aggiornamento del *bucket* di puntatori
 - Se $K \notin NF$, allora K viene inserito rispettando l'ordine:
 - Indice primario: inserimento di un puntatore alla tupla con valore K della chiave;
 - Indice secondario: inserimento di un nuovo *bucket* di puntatori contenente il puntatore alla tupla con valore K della chiave.

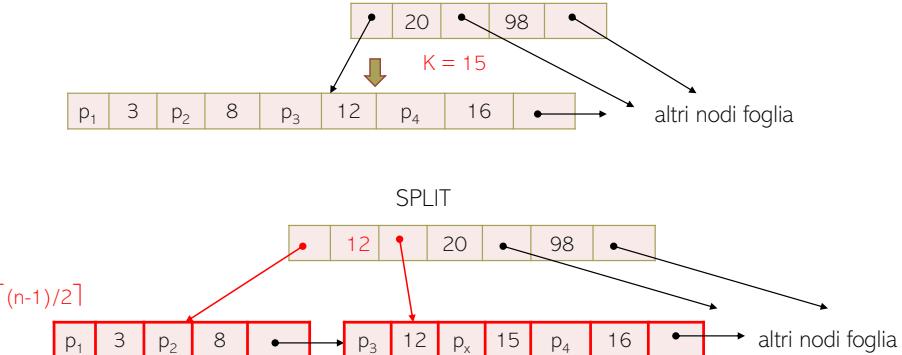


Esiste anche un'altra possibilità, ovvero quello in cui **non è possibile inserire** K in NF e dunque è necessario eseguire uno **split del nodo foglia NF**.

Lo **split di un nodo foglia** consiste nel eseguire una serie di operazioni così che si possa ottenere due nodi, ovvero uno split:

1. Creazione di due nodi foglia;
2. Inserimento dei primi $\lceil \frac{(n-1)}{2} \rceil$ valori nel primo;
3. Inserimento dei rimanenti nel secondo;
4. Inserimento nel nodo padre di un nuovo puntatore per il secondo nodo foglia generato e correzione dei valori chiave presenti nel nodo padre;
5. (Caso estremo) Se il nodo padre è pieno, ovvero ci sono già n puntatori, allora lo SPLIT si propaga al padre e così via fino alla radice, se è necessario.

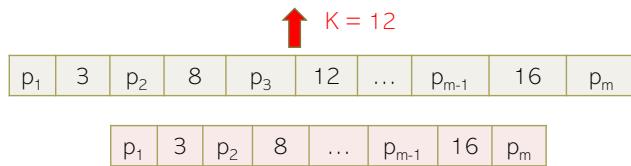
Un **esempio** di SPLIT di un nodo foglia con *fan-out* uguale a 5:



10.2.3 Cancellazione con chiave K

I passi per eseguire una **cancellazione con chiave K** sono i seguenti:

1. Ricerca del nodo foglia NF in cui il valore K deve essere cancellato;
2. Una volta trovato, eliminazione di K dal nodo foglia NF insieme al suo puntatore:
 - L'indice primario viene eliminato direttamente;
 - L'indice secondario esegue prima una *detach* dei puntatori dal *bucket* e poi viene eliminato.



Anche in questo caso, può accadere che dopo la cancellazione di K dal nodo foglia NF, viene violato il vincolo di riempimento minimo (paragrafo 10.1.4 e l'esempio nel paragrafo 10.1.5). In tal caso, il sistema esegue una **MERGE** (pagina successiva) del nodo foglia così da aumentare la dimensione.

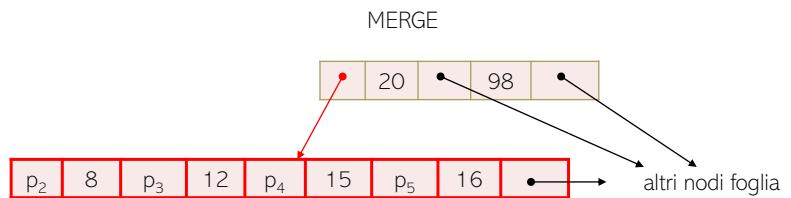
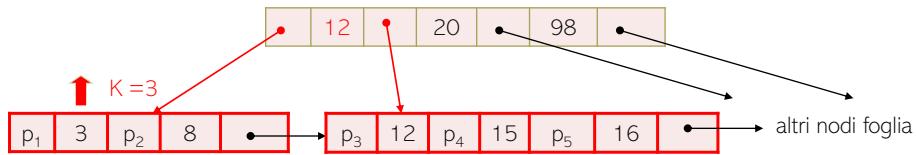
Il **MERGE** di un nodo foglia viene eseguito se nel nodo da unire esistono:

$$\left\lceil \frac{(n-1)}{2} \right\rceil - 1$$

valori chiave. Allora vengono eseguiti i seguenti passaggi:

1. Individuazione del nodo fratello adiacente da unire al nodo corrente;
2. Se i due nodi hanno al massimo $n - 1$ valori chiave:
 - (a) Generazione di un unico nodo contenente tutti i valori;
 - (b) Rimozione di un puntatore dal nodo padre;
 - (c) Correzione dei valori chiave nel nodo padre.
3. Se il punto due era falso, allora vengono distribuiti i valori chiave tra i due nodi e vengono corretti i valori chiave del nodo padre;
4. Nel caso in cui il nodo padre violasse anch'egli il vincolo minimo di riempimento, ovvero meno di $\lceil \frac{n}{2} \rceil$ puntatori presenti, allora l'operazione di MERGE viene propagata al padre e così via fino alla radice, se necessario.

Un **esempio** di applicazione dell'operazione di MERGE:



10.3 Hash

10.3.1 Definizione

Le **strutture ad accesso calcolato**, dette più comunemente **hashing** o **funzioni di hash**, si basano su una funzione di hash che esegue un *mapping* dei valori della chiave di ricerca sugli indirizzi di memorizzazione delle tuple nelle pagine della memoria secondarie:

$$h : K \rightarrow B$$

Con K che rappresenta il dominio delle chiavi e B il dominio degli indirizzi.

Un utilizzo pratico di una funzione di hash negli indici è il seguente:

- Stima del numero di valori che saranno contenuti nella tabella;
- Allocazione di un numero di bucket di puntatori (alias B) uguale al numero stimato;
- Definizione di una funzione di FOLDING che esegue la trasformazione dei valori chiave in numeri interi positivi;

$$f : K \rightarrow Z^+$$

- Definizione di una funzione di HASHING:

$$h : Z^+ \rightarrow B$$

Una funzione di hashing buona è tale quando i **valori** della chiave vengono **distribuiti** in maniera **uniforme ma casuale** all'interno dei *bucket*. Inoltre, è buona norma decidere con accuratezza tale funzione, poiché un suo cambiamento vorrebbe dire decodificare l'intera struttura per poi ricostruire tutto da capo (grave perdita di tempo).

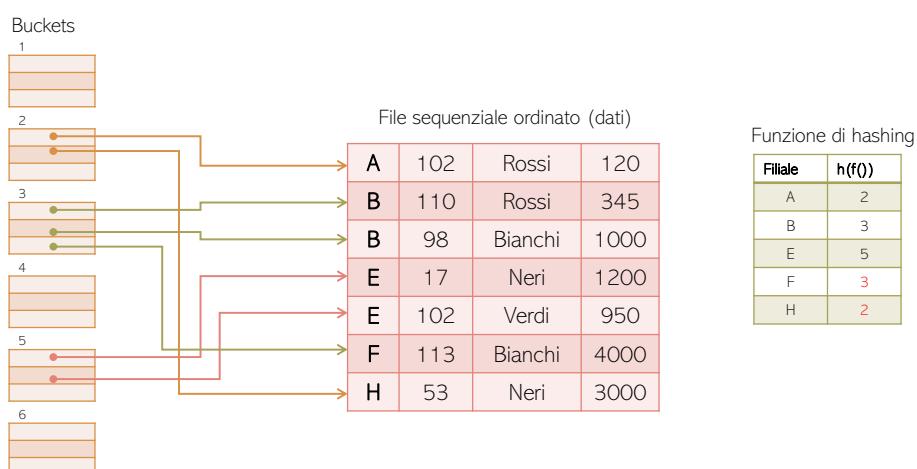


Figura 39: Un **esempio** banale di hashing.

10.3.2 Operazioni

Le **operazioni** eseguite tramite l'hashing sono principalmente tre: la **ricerca**, **l'inserimento** e la **cancellazione**.

La **ricerca** viene eseguita tramite la classica chiave K:

1. Viene calcolata la funzione di hashing $b = h(f(K))$ (costo nullo)
2. Accesso al *bucket* b (costo pari ad uno per pagina)
3. Accesso alle n tuple attraverso i puntatori del *bucket* (costo pari ad m accessi a pagina con $m \leq n$)

Analogamente, sia l'**inserimento** e l'**eliminazione**.

10.3.3 Problema della collisione

Un grave problema da gestire durante l'utilizzo delle funzioni di hash è quello della gestione del **problema della collisione**.

La struttura ad accesso calcolato (hashing) funziona se i *buckets* conservano un basso coefficiente di riempimento. Il problema della collisione si **manifesta** quando dati due valori di chiave $K1$ e $K2$ con $K1 \neq K2$, risulta:

$$h(f(K1)) = h(f(K2))$$

Banalmente, **quando con due valori di chiave diversi, le funzioni di hash coincidono**. Questo è assolutamente da evitare, poiché un **numero eccessivo di collisioni porta alla saturazione del bucket** corrispondente.

La **probabilità che un bucket riceva t chiavi su n inserimenti** è data dalla seguente formula:

$$p(t) = \binom{n}{t} \left(\frac{1}{B}\right)^t \left(1 - \frac{1}{B}\right)^{(n-t)}$$

Dove la prima espressione all'interno delle parentesi indica il numero di combinazione di n oggetti presi ad un passo di t e B indica il numero totale di *buckets*.

La **probabilità di avere più di F collisioni**, dove F rappresenta il numero di puntatori nel *bucket*, è la seguente:

$$p_K = 1 - \sum_{i=0}^F p(i)$$

Una soluzione alle collisioni è la possibilità di adottare una politica di **gestione delle collisioni**. In altre parole, il sistema prevede la possibilità di allocare *bucket* di *overflow* collegati al *bucket* di base. Questo comporta una **riduzione delle prestazioni durante la ricerca** poiché potrebbe essere necessario accedere anche al *bucket* di *overflow*.

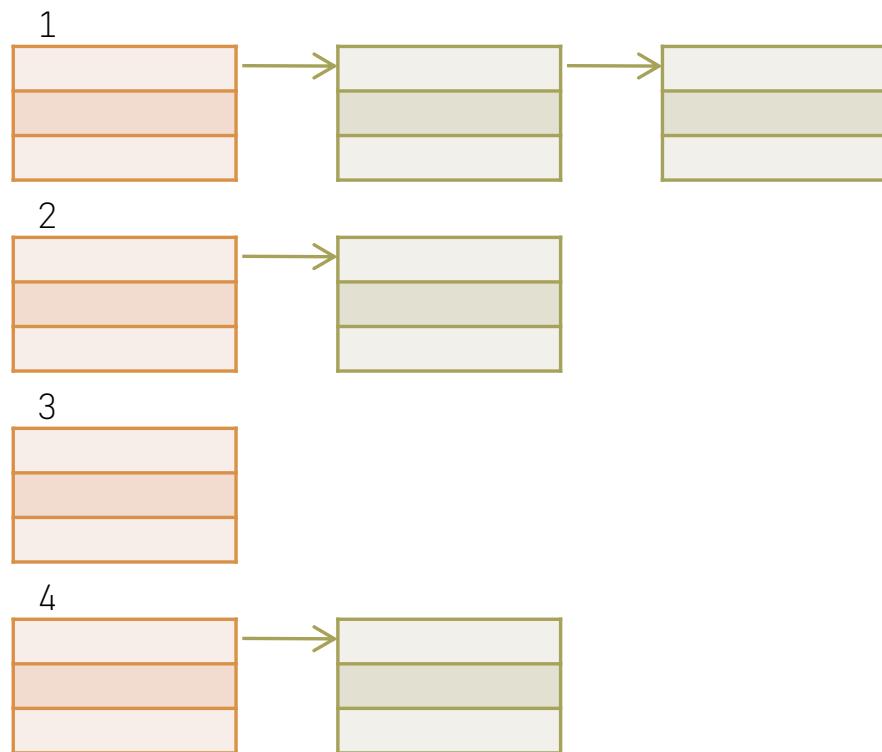


Figura 40: Gestione delle collisioni tramite il supporto di *bucket* di *overflow*.

10.4 Confronto alberi B+ e Hashing

Il confronto verrà effettuato sulle operazioni, quindi sulla ricerca, inserimento e cancellazione.

10.4.1 Operazione di ricerca

Date le selezioni basate su condizioni di uguali, del tipo $A = cost$, si ha:

- **Hashing**, senza *overflow*, tempo costante
- **B+-tree** tempo logaritmo nel numero di chiavi

Mentre, per le selezioni basate su intervalli (*range*) $A > cost1 \wedge A < cost2$:

- **Hashing**, numero elevato di selezioni su condizioni di uguaglianza per scandire tutti i valori del range
- **B+-tree**, tempo logaritmico per accedere al primo valore dell'intervallo, scansione dei nodi foglia (grazie all'ultimo puntatore) fino all'ultimo valore compreso nel range.

10.4.2 Operazione di inserimento e cancellazione

- **Hashing**, tempo costante sommato alla gestione dell'*overflow*
- **B+-tree**, tempo logaritmico nel numero di chiavi sommato alle operazioni di split e/o merge

11 Controllo di concorrenza

11.1 Anomalie delle transazioni concorrenti

L'unità di misura solitamente utilizzata per caratterizzare il carico applicativo di un DBMS è il **numero di transazioni al secondo**, abbreviato in *tps* (*transactions per second*) che devono essere gestite dal DBMS per soddisfare le applicazioni.

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

11.1.1 Perdita di aggiornamento

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

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

- $r_i(x)$ rappresenta la lettura del generico oggetto x
- $w_i(x)$ rappresenta la scrittura del generico oggetto x
- t_1 rappresenta la transazione che esegue r o w
- x si suppone sia uguale a 2 all'inizio
- $x = x + 1$ rappresenta l'incremento effettuato da un programma applicativo, per esempio un update in SQL

L'esecuzione in **sequenza** delle due transazioni ha come risultato il valore 4 all'interno di x .

La **perdita di aggiornamento** (*lost update*) si verifica nel seguente caso tramite l'esecuzione concorrente delle due transazioni:

Tempo	Transazione t_1	Transazione t_2
0	$r_1(x)$	
1	$x = x + 1$	
2		$r_2(x)$
3		$x = x + 1$
4		$w_2(x)$
5		commit
6	$w_1(x)$	
7	commit	

Il valore finale di x è pari a 3, poiché entrambe le transazioni leggono 2 come valore iniziale di x (lettura di t_1 al tempo 0 e lettura di t_2 al tempo 2). Quindi, la perdita di aggiornamento si verifica quando gli **effetti della transazione t_2** , la prima che scrive il nuovo valore di x , **vengono persi**.

11.1.2 Lettura sporca

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

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

- $r_i(x)$ rappresenta la lettura del generico oggetto x
- $w_i(x)$ rappresenta la scrittura del generico oggetto x
- t_1 rappresenta la transazione che esegue r o w
- x si suppone sia uguale a 2 all'inizio
- $x = x + 1$ rappresenta l'incremento effettuato da un programma applicativo, per esempio un `update` in SQL

L'esecuzione in **sequenza** delle due transazioni ha come risultato il valore 4 all'interno di x .

La **lettura sporca** (*dirty read*) si verifica nel seguente caso tramite l'esecuzione concorrente delle due transazioni:

Tempo	Transazione t_1	Transazione t_2
0	$r_1(x)$	
1	$x = x + 1$	
2	$w_1(x)$	
3		$r_2(x)$
4		commit
5	abort	

Il valore finale di x è pari a 2, nonostante la seconda transazione ha letto (tempo 3), e potenzialmente comunicato all'esterno, il valore 3. L'**aspetto critico** di questa anomalia è la lettura della transazione t_2 , la quale vede uno stato intermedio generato dalla transazione t_1 . Tuttavia, t_2 non avrebbe dovuto vedere tale stato, perché successivamente viene eseguito un `abort`.

Il nome di questa anomalia deriva dal fatto che **viene letto un dato che rappresenta uno stato intermedio nell'evoluzione di una transazione**. L'unico modo per ripristinare la correttezza, sarebbe imporre un `abort` anche nella transazione t_2 e, ricorsivamente, di tutte le transazioni che avessero letto dati modificati da t_2 . Purtroppo, questa pratica, oltre ad essere molto onerosa, talvolta è anche non praticabile a causa dell'eventuale comunicazione all'esterno del risultato.

11.1.3 Letture inconsistenti

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

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

- $r_i(x)$ rappresenta la lettura del generico oggetto x
- $w_i(x)$ rappresenta la scrittura del generico oggetto x
- t_1 rappresenta la transazione che esegue r o w
- x si suppone sia uguale a 2 all'inizio
- $x = x + 1$ rappresenta l'incremento effettuato da un programma applicativo, per esempio un `update` in SQL

L'esecuzione in **sequenza** delle due transazioni ha come risultato il valore 4 all'interno di x .

La **lettura inconsistente** si verifica nel seguente caso tramite l'esecuzione concorrente delle due transazioni:

Tempo	Transazione t_1	Transazione t_2
0	$r_1(x)$	
1		$r_2(x)$
2		$x = x + 1$
3		$w_2(x)$
4		commit
5	$r_1(x)$	
6	abort	

In questo caso, la x assume il valore 2 dopo la prima operazione di lettura e il valore 3 dopo la seconda operazione di lettura. La lettura inconsistente si verifica poiché una **transazione che accede due o più volte alla base di dati, trova due o più volte un valore diverso per ciascun dato letto**.

Invece, è opportuno che una transazione che accede due volte alla base di dati trovi esattamente lo stesso valore per ciascun dato letto, e non risenta dell'effetto di altre transazioni.

11.1.4 Aggiornamento fantasma

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

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

- $r_i(x)$ rappresenta la lettura del generico oggetto x
- $w_i(x)$ rappresenta la scrittura del generico oggetto x
- t_1 rappresenta la transazione che esegue r o w
- x si suppone sia uguale a 2 all'inizio
- $x = x + 1$ rappresenta l'incremento effettuato da un programma applicativo, per esempio un `update` in SQL

L'esecuzione in **sequenza** delle due transazioni ha come risultato il valore 4 all'interno di x .

L'**aggiornamento fantasma** (*ghost update*) si verifica nel seguente caso tramite l'esecuzione concorrente delle due transazioni supponendo anche che la base di dati abbia 3 oggetti x, y, z che soddisfano il vincolo di integrità $x + y + z = 1000$:

Tempo	Transazione t_1	Transazione t_2
0	$r_1(x)$	
1		$r_2(y)$
2	$r_1(y)$	
3		$y = y - 100$
4		$r_2(z)$
5		$z = z + 100$
6		$w_2(y)$
7		$w_2(z)$
8		commit
9	$r_1(z)$	
10		$s = x + y + z$
11		commit

La transazione t_2 non altera la somma dei valori e quindi non viola il vincolo di integrità. Tuttavia, la variabile s della transazione t_1 , che dovrebbe contenere la somma di x, y, z , contiene in effetti al termine dell'esecuzione il valore 1100.

In altri termini, la **transazione t_1 osserva solo in parte gli effetti** (di un'altra) della **transazione t_2** , e quindi **osserva uno stato che non soddisfa i vincoli di integrità**.

11.2 Teoria del controllo di concorrenza

Uno **schedule** rappresenta la **sequenza di operazioni di ingresso/uscita presentate da transazioni concorrenti**. Uno schedule S_1 solitamente ha una sequenza del tipo:

$$S_1 : r_1(x) r_2(z) w_1(x) w_2(z) \dots$$

- $r_1(x)$ rappresenta la **lettura** dell'oggetto x effettuata dalla transazione t_1
- $w_2(z)$ rappresenta la **scrittura** dell'oggetto z effettuata dalla transazione t_2

Le operazioni compaiono nello schedule seguendo l'ordine temporale con cui sono eseguite sulla base di dati.

Il **controllo di concorrenza** ha la funzione di **accettare alcuni schedule e rifiutarne altri**, in modo per esempio di evitare che si verifichino le anomalie illustrate nel paragrafo precedente (11.1). Per questo motivo, il modulo che gestisce il controllo di concorrenza viene chiamato **scheduler**: esso ha l'**obiettivo di intercettare le operazioni compiute** sulla base di dati dalle transazioni, decidendo per ciascuna se **rifiutarla o accettarla**.

Uno schedule viene detto **commit-proiezione** quando esso assume che le **transazioni che compaiono nello schedule abbiano un esito noto** (commit o abort). Quindi, **vengono ignorate le transazioni che producono un abort**, togliendo dallo schedule tutte le loro azioni, e concentrandosi solo sulle transazioni che producono un commit.

Questa assunzione viene fatta solo a fini teorici poiché nella realtà uno scheduler deve decidere se accettare o rifiutare le azioni di una transazione senza conoscere il suo esito finale.

Inoltre, uno schedule si definisce **seriale** se **tutte le azioni di ogni transazione compaiono in sequenza, senza essere inframmezzate da istruzioni di altre transazioni**. Per **esempio**, lo schedule S_2 è seriale e vengono eseguite in sequenza le transazioni t_0, t_1, t_2 :

$$S_2 : r_0(x) r_0(y) w_0(x) r_1(y) r_1(x) w_1(y) r_2(x) r_2(y) r_2(z) w_2(z)$$

Uno schedule S_i viene identificato come **serializzabile** se la sua **esecuzione è corretta quando produce lo stesso risultato prodotto da un qualunque schedule seriale S_j delle stesse transazioni**.

11.2.1 View-equivalenza

Si definisce una relazione che lega coppie di operazioni di lettura e scrittura:

- Un'operazione di **lettura** $r_i(x)$ **legge da** una scrittura $w_j(x)$ quando $w_j(x)$ precede $r_i(x)$ e non vi è alcun $w_k(x)$ compreso tra le due operazioni.
- Un'operazione di **scrittura** $w_i(x)$ viene detta una **scrittura finale** se è l'**ultima scrittura** dell'oggetto x che appare nello schedule.
- Due **schedule** vengono detti **view-equivivalenti** ($S_i \approx_V S_j$) se possiedono la stessa relazione “legge da” e le stesse scritture finali.
- Uno **schedule** viene detto **view-serializzabile** se esiste uno **schedule** seriale **view-equivaleente** ad esso. L'insieme degli **schedule** **view-serializzabili** è **VSR**.

Per **esempio** si considerino gli schedule seguenti:

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

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

$$S_5 : w_0(x) r_1(x) w_1(x) r_2(x) w_1(z)$$

$$S_6 : w_0(x) r_1(x) w_1(x) w_1(z) r_2(x)$$

- S_3 è view-equivaleente allo schedule seriale S_4 e quindi è anche view-serializzabile;
- S_5 non è invece view-equivaleente a S_4 , ma è view-equivaleente allo schedule seriale S_6 , e quindi anche view-serializzabile.

È interessante notare che i seguenti schedule corrispondono ad anomalie di perdita di aggiornamento (paragrafo 11.1.1), letture inconsistenti (paragrafo 11.1.3) e aggiornamento fantasma (paragrafo 11.1.4). Nessuno è view-serializzabile:

Perdita di aggiornamento $S_7 : r_1(x) r_2(x) w_2(x) w_1(x)$

Letture inconsistenti $S_8 : r_1(x) r_2(x) w_2(x) r_1(x)$

Aggiornamento fantasma $S_9 : r_1(x) r_1(y) r_2(z) r_2(y) w_2(y) w_2(z) r_1(z)$

Determinare la **view-equivalenza di due schedule** è un problema con **complessità lineare**. Determinare se uno schedule è view-equivaleente a un qualsiasi schedule seriale è un problema NP-difficile⁵, perché può esistere un numero esponenziale di schedule seriali (tutte le permutazioni delle transazioni).

⁵Definizione di un problema NP-difficile: [link Wikipedia](#)

11.2.2 Conflict-equivalenza

Si definisce una nozione di equivalenza più semplice rispetto a quella utilizzata nella view:

- Date due azioni a_i e a_j eseguite da transazioni diverse, quindi con $i \neq j$, si dice che a_i è in **confitto** con a_j se esse **operano sullo stesso oggetto** e almeno una di esse è una **scrittura**.

I conflitti esistenti sono:

- Lettura-scrittura (rw)
- Scrittura-lettura (wr)
- Scrittura-scrittura (ww)

- Lo schedule S_i è **conflict-equivale** allo schedule S_j ($S_i \approx_C S_j$) se i due schedule presentano le stesse operazioni e ogni coppia di operazioni in conflitto è nello stesso ordine nei due schedule.
- Uno schedule risulta **conflict-serializzabile** se esiste uno schedule seriale a esso conflict-equivale. L'insieme degli schedule conflict-serializzabili si chiama **CSR**.

È dimostrato che l'insieme degli schedule CSR è strettamente incluso nell'insieme degli schedule VSR:

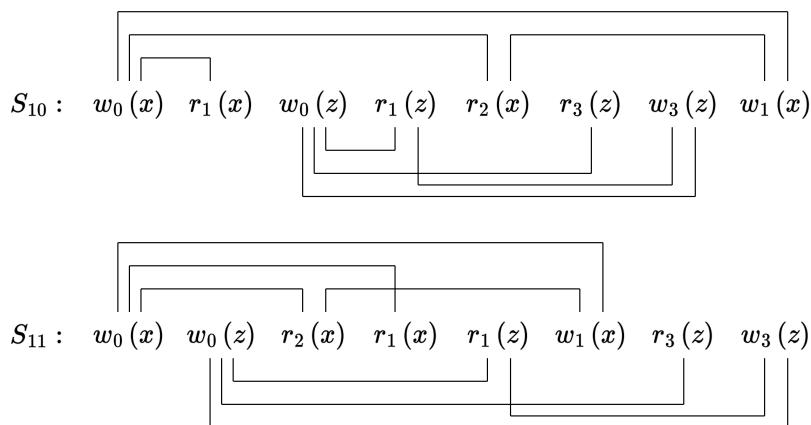
$$CSR \subset VSR$$

Ovvero, esistono schedule che appartengono a VSR ma non a CSR, mentre tutti gli schedule CSR appartengono a VSR.

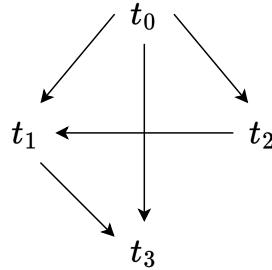
A differenza della view-equivalenza che necessita di verificare tutte le permutazioni prima di dire se uno schedule è view-serializzabile, il conflict-equivalenza può determinare se uno schedule è conflict-serializzabile tramite il **grafo dei conflitti**.

Il grafo è costruito facendo corrispondere un nodo a ogni transazione. Si traccia quindi un arco orientato da t_i a t_j se esiste almeno un conflitto tra un'azione a_i e un'azione a_j e si ha che a_i precede a_j .

Per **esempio**:



Il relativo grafo:



È possibile dimostrare che uno **schedule** è in **CSR** se e solo se il suo grafo dei conflitti è aciclico⁶:

- Dato uno **schedule** $S \in CSR$, per definizione è conflict-equivalente ad uno **schedule seriale** S_0 .
Sia t_1, t_2, \dots, t_n , la sequenza delle transazioni in S_0 . Dato che lo schedule seriale S_0 ha tutti i conflitti nello stesso ordine dello schedule S , nel grafo S ci possono essere solo archi (i, j) con $i < j$ e quindi il grafo non può avere cicli. Quest'ultima affermazione è vera perché un ciclo richiede almeno un arco (i, j) con $i > j$ che non esiste.
- Dato un **grafo** di S aciclico, allora esiste fra i nodi un “ordinamento topologico”, ovvero una **numerazione dei nodi tale che il grafo contiene solo archi (i, j) con $i < j$** .
Lo schedule seriale le cui transazioni sono ordinate secondo l’ordinamento topologico è equivalente a S , perché per ogni conflitto (i, j) si ha sempre $i < j$.

L’**analisi di ciclicità di un grafo ha una complessità lineare** rispetto alla dimensione del grafo stesso.

⁶Per grafo aciclico si intende: [link Wikipedia](#)

11.2.3 Locking a due fasi (2PL)

Esiste un meccanismo che supera i problemi riscontrati nelle due tecniche precedenti (conflict-equivalenza e view-equivalenza). Tale tecnica prende il nome di **locking**, la quale si basa sulla seguente idea: **tutte le operazioni di lettura e scrittura devono essere protette tramite l'esecuzione di opportune primitive (r_lock, w_lock, unlock); lo scheduler (lock manager) riceve una sequenza di richieste di esecuzione di queste primitive da parte delle transazioni, e ne determina la correttezza con una semplice ispezione di una struttura dati.**

1. Ogni **operazione di lettura** deve essere preceduta da un **r_lock** e seguita da un **unlock**;

Il **lock** si dice in questo caso **condiviso**, perché su un dato possono essere contemporaneamente attivi più lock di questo tipo.

2. Ogni **operazione di scrittura** deve essere preceduta da un **w_lock** e seguita da un **unlock**;

Il **lock** si dice in tal caso **esclusivo**, perché non può coesistere con altri lock (esclusivi o condivisi) sullo stesso dato.

Se vengono rispettate queste regole, allora una **transazione è ben formata rispetto al locking**.

Si forniscono alcuni termini per delle azioni:

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

Una **richiesta di lock** è caratterizzata solo dall'**identificativo della transazione** che fa la richiesta, e dalla **risorsa** per la quale la richiesta viene effettuata.

Il lock manager basa le sue decisioni in base alla **tabella dei conflitti**:

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

Tabella 1: La tabella dei conflitti per il metodo di locking.

Alcune precisazioni e caratteristiche della tabella:

- Le ***righe*** identificano le **richieste**;
- Le **colonne** identificano lo **stato della risorsa** richiesta;
- Il **valore di sinistra** (OK, No) identifica l'**esito della richiesta**. In particolare, il valore No rappresenta un conflitto che si può presentare quando:
 - Viene richiesta una lettura/scrittura su un oggetto già bloccato in scrittura;
 - Viene richiesta una scrittura su un oggetto già bloccato in lettura.
- Il **valore di destra** (*r_locked*, *w_locked*, *dipende*, *libero*) identifica lo **stato** che verrà assunto dalla **risorsa dopo l'esecuzione** della primitiva.
- Solo quando un oggetto è bloccato in lettura è possibile dare risposta positiva ad un'altra richiesta di lock in lettura. Da questa caratteristica discende il nome ***lock condiviso*** attribuito al lock in lettura.
- Nel caso di **unlock** di un risorsa bloccata in modo condiviso (lettura), la **risorsa ritorna libera quando non ci sono altre transazioni in lettura che operano su di essa**; altrimenti, essa rimane bloccata in lettura.

Per questo motivo, la colonna *r_locked* assume il valore *dipende* in corrispondenza di *unlock*.

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

Questo provoca una conseguenza durante l'esecuzione, la quale viene divisa in due fasi:

1. (**fase crescente**) Acquisizione dei lock per le risorse a cui si deve accedere.
2. (**fase calante**) I lock acquisiti vengono rilasciati.

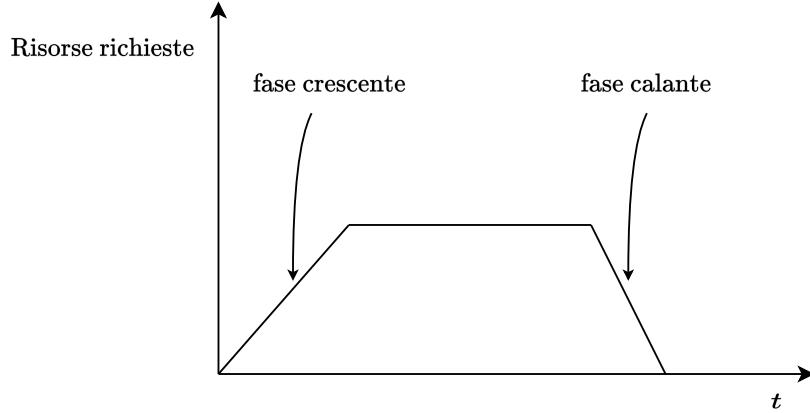


Figura 41: Rappresentazione delle risorse allocate a una transazione con un protocollo di lock a due fasi. L'ascissa rappresenta il tempo, mentre l'ordinata rappresenta il numero di risorse ottenute da una transazione durante la sua esecuzione.

Dimostrazione che $2PL \subset CSR$. Si vuole dimostrare che ogni schedule che rispetti i requisiti del protocollo di lock a due fasi risulta anche uno schedule serializzabile rispetto alla conflict-equivalenza.

Si ipotizzi per assurdo che esista uno schedule S tale che:

$$S \in 2PL \wedge S \notin CSR$$

Se S non appartiene a CSR , allora vuol dire che costruendo il grafo delle dipendenze tra le transazioni si ottiene un ciclo del tipo $t_1, t_2, \dots, t_n, t_1$. Nel caso in cui esista un conflitto tra t_1 e t_2 , vuol dire che esiste una risorsa x su cui operano entrambe le transazioni in modo conflittuale. Affinché la transazione t_2 possa procedere, è necessario che la transazione t_1 rilasci il lock su x .

D'altra parte, osservando il conflitto tra t_n e t_1 , è evidente che esiste una risorsa y su cui operano entrambe le transazioni in modo conflittuale. Affinché la transazione t_1 possa procedere, è necessario che la transazione t_1 acquisisca il lock sulla risorsa y , rilasciato da t_n .

Quindi, la transazione t_1 non può essere a due fasi: essa rilascia la risorsa x prima di acquisire la y . Si conclude che le classi $2PL$ e CSR non sono equivalenti, e quindi che $2PL$ è inclusa strettamente in CSR . Un esempio di schedule non in $2PL$ ma in CSR :

$$S_{12} : \quad r_1(x) w_1(x) r_2(x) w_2(x) r_3(y) w_1(y)$$

In questo caso, la transazione t_1 deve cedere un lock esclusivo sulla risorsa x e successivamente richiedere un lock esclusivo sulla risorsa y , risultando pertanto non a due fasi. Viceversa, lo schedule è conflict-serializzabile rispetto alla sequenza: t_3, t_1, t_2 . Quindi, **la classe $2PL$ è strettamente contenuta nella classe CSR .** QED

Qui di seguito si presenta un **esempio** per osservare come il metodo locking a due fasi evita l'insorgere delle anomalie descritte nei paragrafi (11.1) precedenti:

Tempo	Transazioni		Risorse		
	t_1	t_2	x	y	z
0			free	free	free
1	$r_lock_1(x)$			1:read	
2	$r_1(x)$				
3		$w_lock_2(y)$			2:write
4		$r_2(y)$			
5	$r_lock_1(y)$				1:wait
6		$y = y - 100$			
7		$w_lock_2(z)$			2:write
8		$r_2(z)$			
9		$z = z + 100$			
10		$w_2(y)$			
11		$w_2(z)$			
12		commit			
13		$unlock_2(y)$		1:read	
14	$r_1(y)$				
15	$r_lock_1(z)$			1:wait	
16		$unlock_2(z)$		1:read	
14	$r_1(z)$				
15	$s = x + y + z$				
16	commit				
17	$unlock_1(x)$		free		
18	$unlock_1(y)$			free	
19	$unlock_1(z)$				free

- Per ogni risorsa ci sono quattro possibili stati:
 - free stato libero;
 - i:read bloccato in lettura dalla i -esima transazione;
 - i:write bloccato in scrittura dalla i -esima transazione;
 - i:wait esito negativo di una richiesta di lock della i -esima transazione, posta in stato di attesa.
- Le richieste di lock di t_1 relative alle risorse z e x vengono messe in attesa, e la transazione t_1 può procedere solo quanto tali risorse vengono sbloccate da t_2 .

La lettura sporca (paragrafo 11.1.2) può essere risolta grazie al principio di **locking a due fasi stretto** (strict 2PL): i lock di una transazione possono essere rilasciati solo dopo aver correttamente effettuato le operazioni di **commit/abort**.

A causa di questo vincolo, i lock vengono rilasciati solo al termine della transazione, dopo che ciascun dato è stato portato nel suo stato finale. L'esempio nella pagina precedente utilizza il 2PL stretto, poiché le azioni di rilascio dei lock seguono l'azione di **commit**.

Grazie a questo principio, viene reso **impossibile il verificarsi di letture sporche** (11.1.2), poiché viene **impedito l'accesso**, da parte di altre transazioni, a dati scritti da transazioni che ancora non hanno effettuato il **commit**.

Per concludere, il 2PL può essere sfruttato in maniera differente così da ottenere diversi livelli di isolamento. Nella prossima pagina viene presentata una tabella riassuntiva.

Livello di isolamento	Descrizione
Read uncommitted:	La transazione non chiede lock e non osserva nemmeno i lock esclusivi posti da altre transazioni.
	<ul style="list-style-type: none"> ✓ Perdita di aggiornamento; ✗ Lettura sporca; ✗ Letture inconsistenti; ✗ Aggiornamento fantasma; ✗ Inserimento fantasma;
Read committed:	Richiede lock per le letture, rilasciandoli subito dopo, quindi senza 2PL; in questo modo si evitano le letture sporche, ma non le altre anomalie tipiche delle letture.
	<ul style="list-style-type: none"> ✓ Perdita di aggiornamento; ✓ Lettura sporca; ✗ Letture inconsistenti; ✗ Aggiornamento fantasma; ✗ Inserimento fantasma;
Repeatable read:	Applica il 2PL stretto, ma applicando i lock a singole tuple; sono evitate tutte le anomalie ma non l'inserimento fantasma (<i>phantom</i>), perché non è possibile impedire l'inserimento di nuove tuple.
	<ul style="list-style-type: none"> ✓ Perdita di aggiornamento; ✓ Lettura sporca; ✓ Letture inconsistenti; ✓ Aggiornamento fantasma; ✗ Inserimento fantasma;
Serializable:	Applica il 2PL stretto e i lock di predicato, quindi evita tutte le anomalie.
	<ul style="list-style-type: none"> ✓ Perdita di aggiornamento; ✓ Lettura sporca; ✓ Letture inconsistenti; ✓ Aggiornamento fantasma; ✓ Inserimento fantasma;

11.3 Blocco critico

Il **blocco critico** (*deadlock*) costituisce un problema rilevante. **Si spiega il fenomeno tramite un esempio.**

Si supponga di avere una transazione t_1 che deve eseguire le operazioni $r(x)$, $w(y)$, e una seconda transazione t_2 che deve eseguire $r(y)$, $w(x)$. Se viene utilizzato il protocollo di lock a due fasi, è possibile presentare il seguente schedule:

$r_lock_1(x), r_lock_2(y), read_1(x), read_2(y), w_lock_1(y), w_lock_2(x)$

È evidente che nessuna delle transazioni riesce a procedere e il sistema è bloccato. Il problema è dato dal fatto che t_1 è in attesa che si liberi l'oggetto y , che è bloccato da t_2 , e a sua volta t_2 è in attesa dell'oggetto x , bloccato da t_1 .

Si valuta la **probabilità** che si verifichi un evento del genere. Si consideri una tabella che consiste di n diverse tuple, con identica probabilità di accesso:

- La probabilità che due transazioni che operano un solo accesso vadano in conflitto è $\frac{1}{n}$;
- La probabilità che si verifichi un blocco critico di lunghezza 2 è pari alla probabilità di un secondo conflitto, e quindi vale $\frac{1}{n^2}$.

Le **tecniche utilizzate per risolvere il problema** sono tre:

1. Timeout;
 2. Prevenzione (*deadlock prevention*);
 3. Rilevamento (*deadlock detection*).
-

11.3.1 Timeout

Il **timeout** è una tecnica molto semplice. Le **transazioni rimangono in attesa** di una risorsa **per un tempo** prefissato. Una volta passato questo tempo, se la **risorsa non è stata ancora concessa, la richiesta di lock viene rifiutata**.

In questo modo, nel caso in cui una transazione fosse in deadlock, verrebbe tolta dalla condizione di attesa, e presumibilmente abortita.

Il **valore di timeout** può essere scelto in base a due aspetti:

- Valore **elevato** → risolve tardi i blocchi critici, dopo che le transazioni coinvolte nel blocco hanno passato del tempo in attesa;
- Valore **tropppo basso** → corre il rischio di rilevare come blocchi critici anche situazioni in cui una transazione sta aspettando una risorsa senza che vi sia un vero deadlock, uccidendo inutilmente una transazione e sprecando il lavoro già svolto dalla transazione.

11.3.2 Prevenzione (*deadlock prevention*)

Esistono diverse tecniche di prevenzione, qui di seguito ne vengono presentate due tralasciando l'esistenza della più nota e recente che si basa sul *lock upgrade* (ovviamente non spiegato a lezione):

- Questa tecnica prevede di **richiede il lock di tutte le risorse necessarie alla transazione in una sola volta**.

Problema: le transazioni spesso non conoscono a priori le risorse cui vogliono accedere.

- Un'altra tecnica utilizzata si basa sul fatto che le **transazioni acquisiscano un timestamp**, e consiste nel **consentire l'attesa di una transazione t_i su una risorsa acquisita da t_j solamente se vale una determinata relazione di precedenza fra i timestamp di t_i e t_j** (per esempio $i < j$).

Vantaggio: circa il 50% delle richieste che generano un conflitto possono attendere in coda, mentre nel restante 50% dei casi una transazione deve essere uccisa.

Problema: quale transazione uccidere.

Le politiche di scelta della transazione da uccidere (**applicabile solo alla seconda tecnica presentata**) si dividono in **politiche interrompenti (preemptive)**, cioè se è possibile risolvere il conflitto uccidendo la transazione che possiede la risorsa, e **politiche non interrompenti**, ovvero una transazione può essere uccisa solo all'atto di fare una nuova richiesta.

Un **tipo di politica adottata** è la seguente:

- Vengono **uccise le transazioni che hanno fatto meno lavoro**.

Problema: può capitare che una transazione faccia accesso, all'inizio della propria elaborazione, ad un oggetto cui accedono molte altre transazioni. In questo modo, la transazione trova sempre un conflitto, ed essendo la transazione che ha fatto meno lavoro, viene uccisa ripetutamente (**blocco individuale, starvation**).

Soluzione al problema: garantire che ogni transazione non possa essere uccisa un numero illimitato di volte. Tecnica attuabile **mantenendo lo stesso timestamp quando una transazione viene fatta abortire e ripartire, dando nel contempo priorità crescente alle transazioni più “anziane”**.

Quest'ultima soluzione (tecnica) al problema non viene mai usata nei DBMS commerciali, in quanto mediamente si uccide una transazione ogni due conflitti, mentre la probabilità di insorgenza del blocco critico è di gran lunga inferiore alla probabilità di conflitto.

11.3.3 Rilevamento (*deadlock detection*)

La tecnica di **rilevamento** prevede di **non porre vincoli al comportamento del sistema**, ma di **controllare il contenuto delle tabelle di lock** tutte le volte che si ritiene necessario, così da **rilevare eventuali situazioni di blocco**.

Il controllo viene effettuato a intervalli prefissati, o quando scade un timeout di attesa di una transazione.

Il rilevamento di un blocco critico richiede di analizzare le relazioni di attesa tra le varie transazioni e di determinare se esiste un ciclo. La **ricerca di cicli in un grafo risulta abbastanza efficiente**. Per questo motivo, alcuni DBMS commerciali utilizzano questa tecnica.

12 Gestore delle interrogazioni: esecuzione e ottimizzazione

12.1 Definizione

Il **gestore delle interrogazioni** è responsabile dell'esecuzione efficiente di operazioni che sono specificate a livello molto alto. Esso si sviluppa nel seguente modo:

1. Riceve in ingresso un'interrogazione scritta in SQL;
2. L'interrogazione viene analizzata per determinare eventuali errori lessicali, sintattici o semantici. Inoltre, il sistema accede al dizionario dei dati per leggere l'informazione ivi contenuta e consentire così i controlli; dal dizionario dei dati vengono lette anche informazioni statistiche relative alle dimensioni delle tabelle;
3. Accettando l'interrogazione, quest'ultima viene tradotta in una forma interna di tipo algebrico;
4. L'ottimizzazione ha inizio:
 - Ottimizzazione di tipo algebrico, in cui vengono effettuate tutte le trasformazioni algebriche che sono sempre convenienti;
 - Ottimizzazione che dipende sia dalla tipologia dei metodi di accesso ai dati supportati dal sottostante livello sia dal modello dei costi assunto;
 - Generazione del codice, la quale utilizza i metodi di accesso ai dati. Si ottengono quindi programmi di accesso in formato “oggetto” o “interno” che richiedono l'uso delle strutture dati fornite dal sistema.

L'ottimizzazione agisce a tempo di compilazione. Infatti, nel caso in cui l'interrogazione venga compilata una volta ed eseguita molteplici volte, il codice viene prodotto e memorizzato nella base di dati, assieme a un'indicazione delle dipendenze del codice dalle particolari versioni di tabelle e indici della base di dati, descritte nel dizionario dei dati.

Invece, talvolta un'interrogazione viene compilata e immediatamente eseguita, senza essere memorizzata.

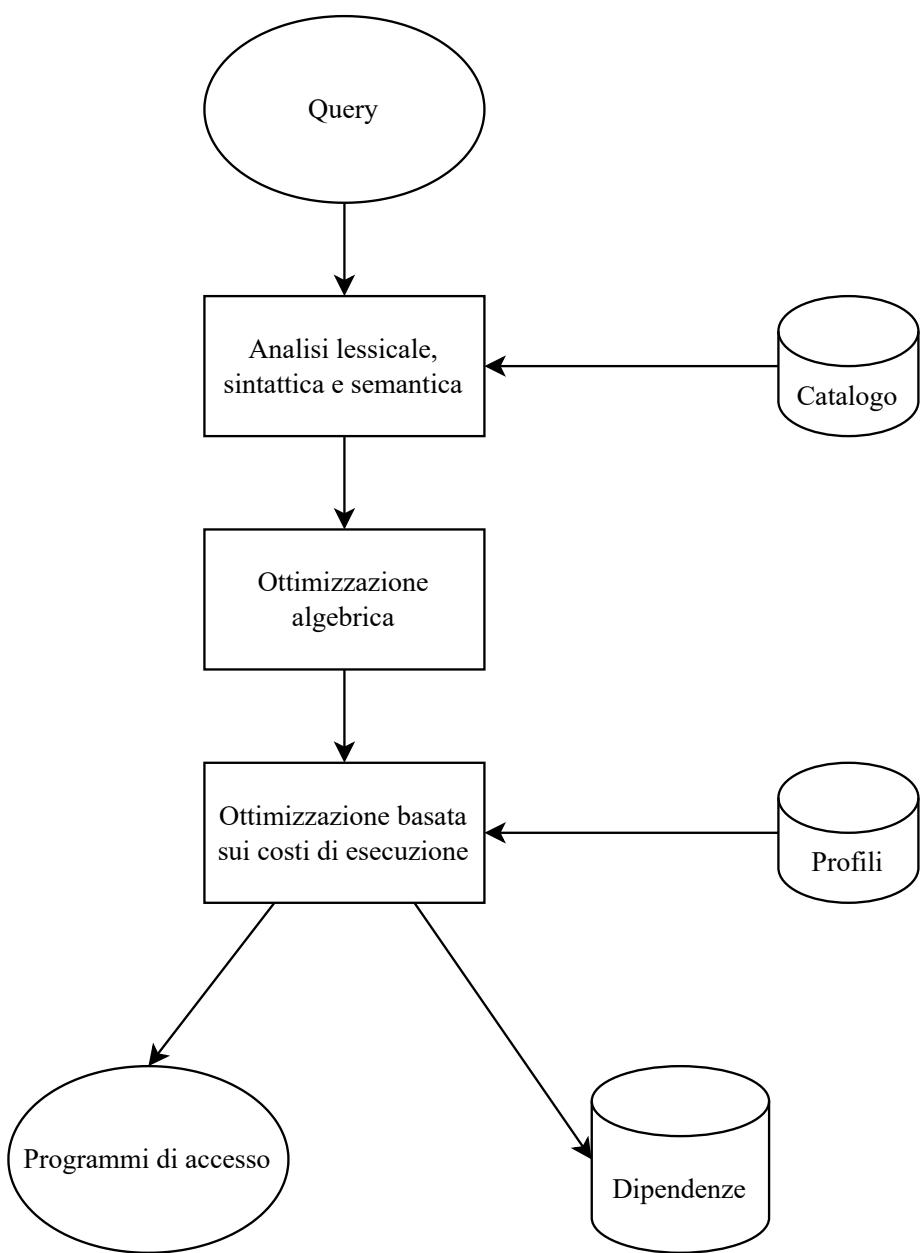


Figura 42: Compilazione di un'interrogazione.

12.2 Profili delle relazioni

Ciascun DBMS possiede informazioni quantitative relative alle caratteristiche delle tabelle, organizzate in strutture dati, dette **profili delle relazioni**, le quali vengono memorizzate nel dizionario dei dati. I profili contengono:

- **Cardinalità** $\text{CARD}(T)$ (numero di tuple) di ciascuna tabella T ;
- **Dimensione** in byte, $\text{SIZE}(T)$, di ciascuna tupla di T ;
- **Dimensione** in byte, $\text{SIZE}(A_j, T)$, di ciascun attributo A_j di T ;
- **Numero di valori distinti**, $\text{VAL}(A_j, T)$, di ciascun attributo A_j di T ;
- **Valore minimo** $\text{MIN}(A_j, T)$ e quello **massimo** $\text{MAX}(A_j, T)$ di ciascun attributo A_j di T .

I profili vengono calcolati in base ai dati effettivamente memorizzati nelle tabelle, utilizzando opportune primitive di sistema.

Nelle valutazioni delle stime di costo si tiene conto del profilo delle relazioni.

12.3 Operazioni di scansione

Un'operazione di **scansione** (*scan*) opera contestualmente varie operazioni di tipo algebrico ed extra-algebrico. Quindi, oltre alla scannerizzazione viene effettuata in combinazione anche una delle seguenti operazioni:

- **Proiezione** su di una lista di attributi;
- **Selezione** su di un predicato;
- **Inserimenti, cancellazioni e modifiche** delle tuple quando vi si fa accesso durante la scansione.

12.4 Ordinamenti

La necessità di operazioni di **ordinamento** emerge principalmente per tre motivi:

1. Ai fini dell'applicazione, poiché si desiderano risultati ordinati;
2. Per una corretta realizzazione delle proiezioni, con eliminazione dei duplicati;
3. Utile per operazioni quali join o di raggruppamento.

Il **problema di ordinare** strutture dati riguarda il corso di algoritmi, tuttavia viene introdotto un **algoritmo utilizzato nella memoria secondaria** chiamato **Z-way Sort-Merge**.

12.4.1 Passaggi dell'algoritmo Z-way Sort-Merge

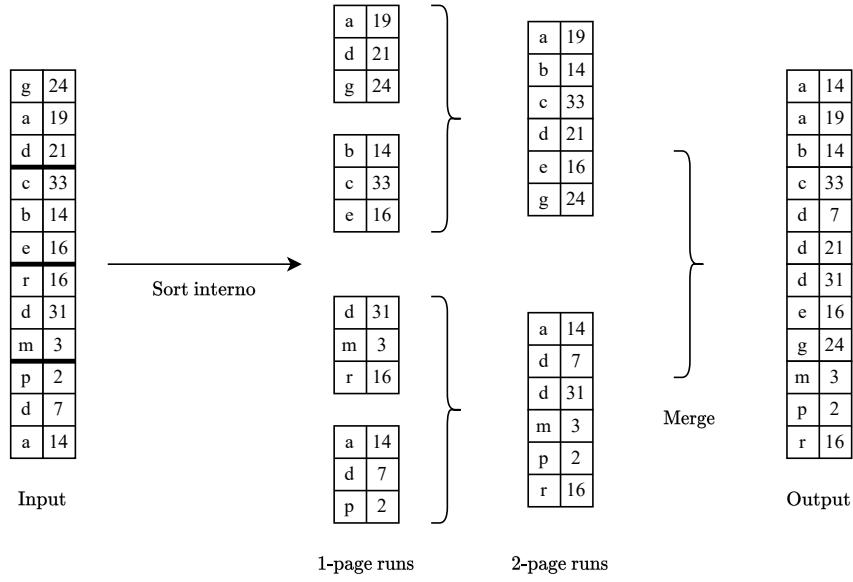
- **Sort interno:**

1. Vengono lette una alla volta le pagine della tabella;
2. Le tuple di ogni pagina vengono ordinate facendo uso di un algoritmo di sort interno (come il QuickSort);
3. Ogni pagina ordinata, chiamata *run*, viene scritta su memoria secondaria in un file temporaneo

- **Merge:** applicando uno o più passi di fusione (a seconda del parametro Z , quindi due passi con 2-way Sort-Merge, tre passi con 3-way Sort-Merge e così via), le *run* vengono unite fino a produrre un'unica pagina (o *run*).

12.4.2 Esempio di applicazione dell'algoritmo

Si supponga di dover ordinare un input che è composto in una tabella di NP pagine e di avere a disposizione solo NB buffer in memoria centrale, tale per cui $NB < NP$. Si considera inoltre il caso base a due vie $Z = 2$ e si supponga di avere a disposizione solo 3 buffer in memoria centrale $NB = 3$.



Vengono prese le pagine in input a coppie di 3 e ordinate dal primo buffer; successivamente a coppie di 6 e ordinate dal secondo buffer; infine, il risultato sarà prodotto dal merge eseguito dal terzo buffer.

Si consideri come **costo** il **numero di accessi alla memoria secondaria**. Allora, nel caso di $Z = 2$ e con $NB = 3$ si possono fare alcune considerazioni. Nella fase di sort interno si leggono/riscrivono NP pagine e ad ogni passo di merge si leggono/riscrivono NP pagine. Quindi, il numero di passi di merge è pari a:

$$\lceil \log_2 (NP) \rceil$$

Poiché ad ogni passo il numero di run si dimezza ($Z = 2$). Il costo complessivo è dunque pari a:

$$2 \cdot NP \cdot (\lceil \log_2 NP \rceil)$$

12.5 Accesso diretto

La tecnica di **accesso diretto** viene utilizzata quando è possibile leggere o scrivere un record **senza** dover necessariamente **esaminare il file in modo sequenziale**, ma è possibile ottenere in altro modo, a partire dal valore di un campo, **l'indirizzo del blocco in cui il record si trova**.

Per eseguire questa tecnica è necessaria la presenza di una struttura *hash* o indice:

- Gli indici favoriscono accessi puntuali, ovvero $A_i = V$, oppure intervalli del tipo $V_1 \leq A_i \leq V_2$. Si dice in questo caso che un predicato dell'interrogazione è **valutabile** tramite l'indice.
- Le strutture *hash* sono molto efficienti per gli accessi puntuali, ma purtroppo non supportano le ricerche per intervallo.

Dopo aver indicato i tipi di struttura necessari per utilizzare l'accesso diretto, si indicano i loro utilizzi in base al numero di predicati e altro:

- L'interrogazione ha **un solo predicato** valutabile, allora è conveniente utilizzare l'indice o la struttura *hash*;
- L'interrogazione ha **una congiunzione di predicati** (e.g. \wedge) valutabili tramite indice o funzione *hash*, il DBMS sceglie il più selettivo dei due per l'accesso diretto;
- L'interrogazione ha **una disgiunzione di predicati** (e.g. \vee), basta che **uno di loro non sia valutabile** per imporre l'uso di una scansione completa;
- Simile al punto precedente, quindi l'interrogazione ha **una disgiunzione di predicati**, ma essi sono **tutti valutabili**, allora è possibile utilizzare gli indici oppure una scansione. Nel caso di utilizzo degli indici, però è necessario eliminare i duplicati di quelle tuple che vengono ritrovate tramite più indici.

12.6 Metodo di join

Il join è l'operazione più gravosa per un DBMS poiché il rischio di un'esplosione del numero di tuple del risultato è alto. Per rendere il più efficiente possibile questa operazione, esistono tre tecniche: **nested loop**, **merge scan**, **hash-based**.

12.6.1 Nested loop

Nel **nested loop** una tabella viene definita come **esterna** e una tabella come **interna**. L'algoritmo si compone dei seguenti passaggi:

1. Viene eseguita una **scansione** sulla **tabella esterna**;
2. Per **ogni tupla** ritrovata dalla scansione, viene **prelevato il valore dell'attributo di join** e vengono **cercate le tuple della tabella interna che hanno lo stesso valore**.
3. A questo punto è utile avere una struttura *hash* o un indice sull'attributo di join della tabella interna, che può essere creato *ad hoc*. Nel caso non fosse possibile sfruttare una di queste strutture, è necessario eseguire una scansione sulla tabella interna per ogni valore di join della tabella esterna.

Questa tecnica prende il nome dal suo *modus operandi* poiché propone una **scansione “nidificata”** nell'altra.

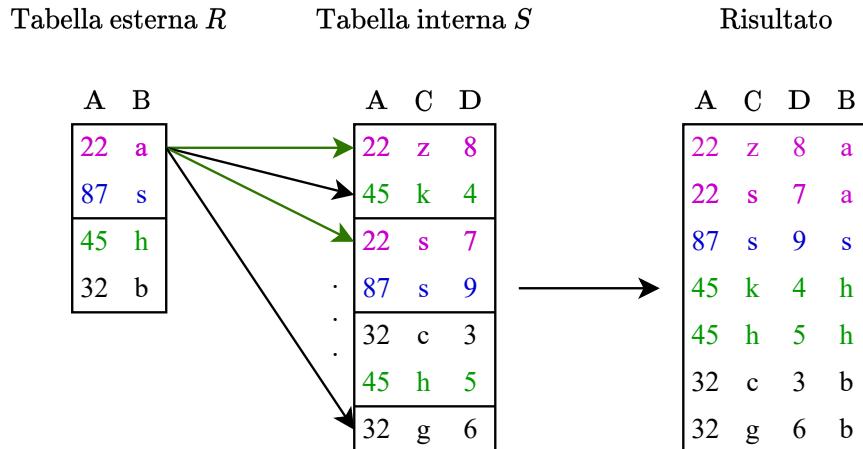


Figura 43: **Esempio** di *nested loop* con una relazione di join del tipo $R.A = S.A$.

Il **costo** dell'algoritmo varia a seconda dell'implementazione e a seconda dello spazio messo a disposizione nei buffer. Si supponga che ci sia 1 buffer a disposizione per ogni tabella (interna ed esterna) e che NR sia il numero di tuple (*row*) e NP il numero di pagine:

- Costo dell'algoritmo generale è

$$NP(Esterna) + NR(Esterna) \times NP(Interna)$$

accessi a memoria secondaria;

- Costo dell'algoritmo con un indice *ad hoc* è

$$NP(Esterna) + NR(Esterna) \times \left(\text{ProfonditàIndice} + \frac{NR(Interna)}{VAL(A, Interna)} \right)$$

dove $VAL(A, Interna)$ rappresenta il numero di valori distinti dell'attributo A che compaiono nella relazione *Interna*.

12.6.2 Merge scan

Il **merge scan** richiede di esaminare le tabelle secondo l'ordine degli attributi di join ed è quindi **particolarmente efficiente quando le tabelle sono già ordinate oppure quando sono definiti su di esse indici adeguati**.

L'**algoritmo** è semplice poiché si basa sulla **scannerizzazione parallela delle due tabelle**, basate sull'ordinamento, **con i classici algoritmi di fusione (merge)**. In questo modo, le scansioni possono ritrovare nelle tuple valori ordinati degli attributi di join; quando coincidono, vengono generate ordinatamente tuple del risultato.

Il **costo** totale dell'algoritmo è il seguente, ricordando NR come il numero di tuple (*row*) e NP come il numero di pagine. Utilizzando gli indici:

$$NP(Esterna) + NR(Interna)$$

12.6.3 Hash-based

La tecnica di **hash-based** si sviluppa in due passi (versione **partizionata**):

1. Data una funzione h di hash sugli attributi di join, essa viene utilizzata per memorizzare una copia di ciascuna delle due tabelle.

Si supponga che h faccia corrispondere i valori del dominio di tali attributi a B partizioni su ciascuna tabella;

2. La costruzione delle tuple del risultato avviene grazie agli stessi valori ottenuti eseguendo l'operazione di hash. Infatti, due valori identici restituiscono esattamente lo stesso hash.

Esiste anche la **versione non partizionata** in cui le tuple vengono calcolate tramite la funzione di *hash* e ricercate direttamente.

Il **costo** è differente a seconda della versione adottata:

- Nel caso della versione senza partizione il costo è:

$$NP(Esterna) + NP(Interna)$$

- Nel caso della versione partizionata:

$$NP(Esterna) \times NR(Interna)$$

13 PostgreSQL

13.1 Introduzione e installazione pgAdmin 4

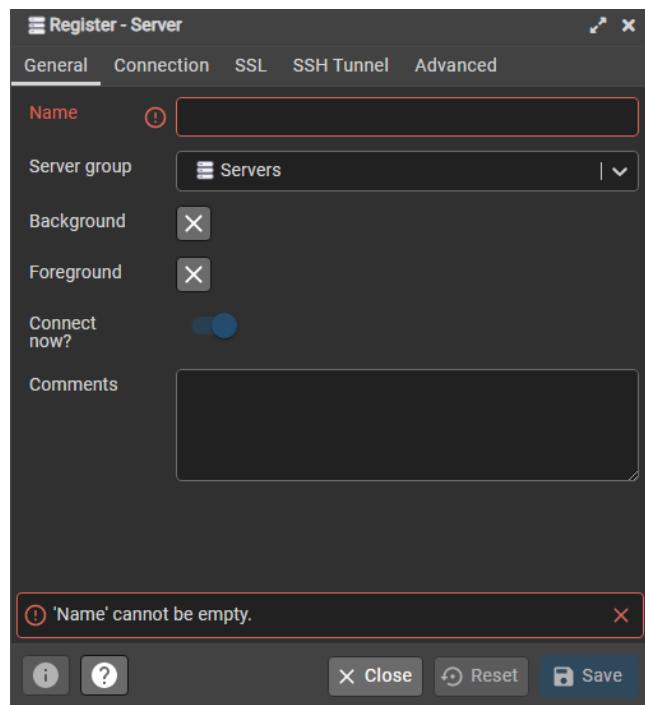
Il sistema PostgreSQL è un Relational Data Base Management System (RDBMS o DBMS) con alcune funzionalità orientate agli oggetti. È possibile installarlo sui sistemi più famosi: Windows, Mac OS, Linux, ecc. Viene gestito da un gruppo di volontari e la sua pagina ufficiale è la seguente: [PostgreSQL](#).

Per installarlo su windows si seguono i seguenti passi:

1. Si scarica il `setup.exe` dalla piattaforma ufficiale: [link download](#);
2. Si esegue il setup andando avanti e lasciando come porta quella predefinita suggerita dal programma;
3. Si rifiuta il download aggiuntivo di un altro programma al termine dell'installazione.

Al termine dell'installazione, è possibile aprire il programma pgAdmin 4, cercandolo banalmente dal menù start di Windows, e iniziare a configurarlo per collegarsi alla piattaforma. La configurazione del server dell'università è la seguente:

1. Sulla colonna di sinistra, si clicca con il destro sopra il nome Server → Register → Server barra della applicazioni si clicca su File → Preferences;
2. All'apertura della seguente finestra:



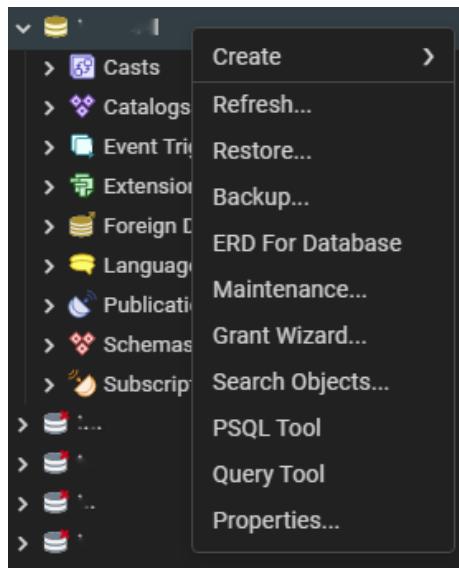
Si inserisce come nome un alias del server, per esempio “gimmy”.
Nella scheda Connection:

- (a) Si inserisce come Host name l’indirizzo del server di base di dati, ovvero: dbserver.scienze.univr.it
- (b) Nel campo port la porta utilizzata durante l’installazione, se è stata lasciata quella di default è la 5432
- (c) Nei campi Maintenance database e Username si inserisce l’username utilizzato per identificarsi sulla piattaforma ESSE3, quindi id123abc
- (d) Il campo password è possibile riempirlo con la password utilizzata per identificarsi sulla piattaforma ESSE3, oppure lasciarlo vuoto per poi scrivere la password al primo accesso al database.

3. Cliccare su Save.

Una volta configurata la piattaforma, sarà possibile accedervi tramite la colonna di sinistra. Sotto la voce Server sarà possibile trovare l’alias indicato nella configurazione e tutti i database presenti all’interno della piattaforma.

Andando sul proprio id, si potrà accedere al database personale. Per eseguire le operazioni di query, si utilizza Query Tool. Per avviarlo basta andare sul proprio database personale → click destro → Query Tool:



Alla sua apertura sarà possibile eseguire query sulla propria base di dati inizialmente vuota. Dopo aver scritto un’espressione ben formattata, sarà possibile eseguirla con la scorciatoia da tastiera F5 o cliccando sul tasto play in alto.

ATTENZIONE! Al termine di ogni sessione, è necessario scollegarsi dal server cliccando con il destro sull’alias iniziale → Disconnect from server.

13.2 Se ti è piaciuto pgAdmin, adorerai VSCode

Su Visual Studio Code esiste un'estensione che consente di eseguire le query. Perché farlo? Esistono alcune persone, come il sottoscritto, che magari preferiscono avere uno strumento un po' più potente/“comodo” rispetto a pgAdmin 4. Da questa necessità nasce questa brevissima introduzione all'installazione dell'estensione PostgreSQL su VSCode.

L'estensione da scaricare è la seguente:

Nome: PostgreSQL

ID: ckolkman.vscode-postgres

Descrizione: PostgreSQL Management Tool

Versione: 1.4.0

Editore: Chris Kolkman

Collegamento di Visual Studio Marketplace: <https://marketplace.visualstudio.com/items?itemName=ckolkman.vscode-postgres>

Una volta scaricata, basterà cliccare sulla figura dell'estensione e cliccare sul tasto + in alto a destra. I dati da inserire in ordine saranno:

1. Il server a cui collegarsi, quindi dbserver.scienze.univr.it
2. La root, quindi l'username utilizzato durante il login su Esse3 (id123ab)
3. La password utilizzata per accedere ad Esse3
4. Il numero di porta (invariato 5432)
5. Standard connection
6. Il database a cui collegarsi, ovvero il vostro id123ab

A questo punto, per eseguire le query basterà cliccare con il tasto destro sopra il vostro database (sarà visibile sulla colonna di sinistra dalla quale è stato cliccato il pulsante +) → New Query. A questo punto scrivendo una query, potrà essere eseguita selezionando il testo da eseguire → click destro → Run Query. Si aprirà una nuova schermata con il risultato.

The screenshot shows the PostgreSQL extension interface in Visual Studio Code. On the left, a context menu is open over a database named 'museo'. The menu items include 'Run Query' (selected), 'F5', 'Cambia tutte le occorrenze (CTRL+F2)', 'Effettua refactoring... (CTRL+MAIUSC+R)', 'Commit Changes >', 'Taglia (CTRL+X)', 'Copia (CTRL+C)', 'Incolla (CTRL+V)', 'Spelling >', 'Aggiungi a espressione di controllo', 'Esegui fino al cursore', and 'Riquadro comandi... (CTRL+MAIUSC+P)'. On the right, a results table titled 'Results: untitled:untitled-1' displays three rows of data. The columns are: titolo (character varying), inizio_date, fine_date, museo (text), città (text), prezzointero (numeric), and prezziordotto (numeric). The data is as follows:

	titolo character varying	inizio_date	fine_date	museo text	città text	prezzointero numeric	prezziordotto numeric
1	PrimoTitolo	Wed Mar 04 2020 00:00:00 GMT+0100 (Ora standard dell'Europa centrale)	Sat Oct 10 2020 00:00:00 GMT+0200 (Ora legale dell'Europa centrale)	PrimoMuseo	PrimaCittà	10.00	5.00
2	SecondoTitolo	Sun Jun 07 2020 00:00:00 GMT+0200 (Ora legale dell'Europa centrale)	Tue Nov 10 2020 00:00:00 GMT+0100 (Ora standard dell'Europa centrale)	SecondaMuseo	SecondaCittà	90.00	5.00
3	TerzoTitolo	Tue Jan 22 2019 00:00:00 GMT+0100 (Ora standard dell'Europa centrale)	Sat Jul 10 2021 00:00:00 GMT+0200 (Ora legale dell'Europa centrale)	TerzoMuseo	TerzaCittà	5.00	5.00

13.3 Caratteri (`character`, `character varying`, `text`)

Il dominio `character` consente di rappresentare singoli caratteri oppure stringhe di lunghezza fissa. Per indicare quest'ultima, viene scritto un valore all'interno delle parentesi tonde dopo il comando. Nel caso in cui non fosse presente, il dominio rappresentare un carattere singolo.

Sintassi:

```
1 character [ ( Lunghezza ) ]
2   [ character set NomeFamigliaCaratteri ]
```

Il campo lunghezza e l'insieme (*set*) sono opzionali. È possibile abbreviare il comando scrivendo `char`.

Esempio di stringa di 20 caratteri: `character(20)`.

Il dominio `character varying` consente di rappresentare stringhe di lunghezza variabile. Il suo funzionamento è identico a quello di `character` con la differenza che la lunghezza è variabile in questo caso.

Sintassi:

```
1 character varying [ ( Lunghezza ) ]
2   [ character set NomeFamigliaCaratteri ]
```

È possibile abbreviare il comando scrivendo `varchar`.

Esempio di stringa di caratteri dell'alfabeto greco a lunghezza variabile, di lunghezza massima 1000: `character varying (1000) character set Greek`.

Il dominio `text` è un'estensione di PostgreSQL e indica una stringa di lunghezza variabile **senza limite** fissato. La sua sintassi è banale: `text`.

13.4 Booleani (`boolean`)

Il dominio `boolean` consente di rappresentare singoli valori booleani: *true* e *false*.

13.5 Tipi numerici esatti (`numeric`, `decimal`, `integer`, `smallint`, `bigint`)

I tipi numerici esatti sono una famiglia contenenti i domini che consentono di rappresentare i valori esatti, interi o con una parte decimale di lunghezza prefissata.

Sintassi:

```
1 numeric [ ( Precisione [, Scala] ) ]
2 decimal [ ( Precisione [, Scala] ) ]
3 bigint
4 integer
5 smallint
```

- `numeric` e `decimal` rappresentano i numeri in base **decimale**. I loro parametri sono:
 - `Precisione`, specifica il numero di cifre significative;
 - `Scala`, specifica la scala di rappresentazione, ovvero si indica quante cifre devono comparire dopo la virgola

La loro **differenza** principale riguarda che la precisione nel dominio `numeric` rappresenta un valore esatto, mentre in `decimal` rappresenta un requisito minimo. Nel caso in cui non venga specificata la dimensione, il sistema usa un valore caratteristico dell'implementazione. Se la scala non si specifica, si assume che valga zero.

- `bigint`, `integer` e `smallint` utilizzano la rappresentazione binaria del calcolatore per rappresentare i valori interi. L'ordine di grandezza dei domini è quello scritto.

Esempi:

- Tutti i numeri decimali con tre valori dopo la virgola e un valore intero (range $-9,999$ e $+9,999$): `decimal(4)`
- I valori con 4 valori dopo la virgola e due interi (range $-99,9999$ e $+99,999$): `decimal(6, 4)`

13.6 Tipi numerici approssimati (`float`, `real`, `double precision`)

Per la rappresentazione di valori reali approssimati, utili nelle rappresentazioni di grandezze fisiche per esempio, si utilizzano principalmente tre comandi.

Sintassi:

```
1 float [ ( Precisione ) ]
2 real
3 double precision
```

Consentono di descrivere numeri approssimati mediante una rappresentazione in virgola mobile, in cui a ciascun numero corrisponde una coppia di valori: mantissa ed esponente. L'importante è sapere che a `float` è possibile assegnare una precisione, la quale rappresenta il numero di cifre dedicate alla rappresentazione della mantissa.

13.7 Istanti e intervalli temporali (`date`, `time`, `timestamp`, `interval`)

Per descrivere **intervalli temporali** o **istanti temporali**, si utilizzano diverse forme.

Sintassi:

```
1 date
2 time [ (Precisione) ] [ with time zone ]
3 timestamp [ (Precisione) ] [ with time zone ]
4 interval PrimaUnitaDiTempo [ to UltimaUnitaDiTempo ]
```

Il campo:

- `date` ammette i campi `year`, `month`, `day`
- `time` ammette i campi `hour`, `minute`, `second`
- `timestamp` ammette tutti i campi elencati

Nei due campi `time` e `timestamp` è possibile accedere al fuso orario tramite `with time zone`. Quindi, si accede ai due campi `timezone_hour` e `timezone_minute` che rappresentano la differenza di fuso orario tra l'ora locale a l'ora universale (UTC).

L'intervallo di tempo rappresenta la durata di un evento e può prendere i seguenti parametri:

- `interval year to month` per indicare che la durata dell'intervallo di tempo deve essere misurata in numero di anni e di mesi. Esempio di rappresentazione di intervalli con numero di anni a cinque cifre (range fino a 99'999 e 11 mesi): `interval year(5) to month`
- `interval day to second` per indicare che la durata dell'intervallo di tempo deve essere misurata in numero di giorni e di secondi. Esempio di rappresentazione di intervalli con numero di giorni a quattro cifre e secondi a sei cifre (range fino a 9'999 giorni, 23 ore, 59 minuti, 59, 999999 secondi): `interval day(4) to second(6)`

Attenzione! Non è possibile mischiare gli insiemi (e.g. `interval year to second`) poiché questo vorrebbe dire sovraccaricare il sistema di calcolo. Dunque, questa operazione non è ammessa.

13.8 Oggetti di grandi dimensioni (`blob`, `clob`)

I due domini `blob`, `clob` consentono di rappresentare oggetti molto grandi, come immagini o video, costituiti da una sequenza arbitraria di valori binari (`blob`, *binary large object*) o di caratteri (`clob`, *character large object*). Il sistema garantisce solo la loro memorizzazione, ma non consente di utilizzarli come criterio di selezione per le interrogazioni.

13.9 Definizione delle tabelle (`create table`)

Una tabella `table` SQL è costituita da una collezione ordinata di attributi e (optional) da un insieme di vincoli.

Sintassi:

```
1 create table NomeTabella
2 ( NomeAttributo Dominio [ ValoreDiDefault ] [ Vincoli ]
3   { , NomeAttributo Dominio [ ValoreDiDefault ] [ Vincoli ] }
4   AltriVincoli
5 )
```

Un esempio della creazione di una tabella:

```
1 create table Dipartimento
2 (
3   Nome      varchar(20) primary key,
4   Indirizzo varchar(50),
5   Citta     varchar(20)
6 )
```

Attenzione alla formattazione!

13.10 Definizione dei domini (create domain)

Nella definizione precedente di tabella, si può far riferimento ai domini predefiniti del linguaggio o a domini definiti dall'utente a partire dai domini predefiniti. Ovvero, è possibile creare insiemi di valori che formano un dominio.

Sintassi:

```
1 create domain NomeDominio as TipoDiDati  
2   [ ValoreDiDefault ]  
3   [ Vincolo ]
```

Il dominio è caratterizzato dal proprio nome, da un valore di default e da un insieme di vincoli eventuali.

Esempio:

```
1 create domain giorniSettimana as char (3)  
2   check(value in('LUN','MAR','MER','GIO','VEN','SAB','DOM'));
```

13.11 Valori di default (default)

I valori di default, come si è visto nei precedenti paragrafi, sono valori che una riga di una tabella assume nel momento in cui viene aggiunta all'interno della base di dati senza che tale campo abbia ricevuto una modifica. Quando il valore di default non viene specificato, si assume il valore nullo.

Sintassi:

```
1 default < GenericoValore | user | null >
```

Si deve scegliere solo uno dei tre valori:

- GenericoValore, rappresenta un valore compatibile con il dominio
- user, impone come valore di default l'identificativo dell'utente che esegue il comando di aggiornamento della tabella
- null, valore nullo

13.12 Vincoli intrarelazionali

13.12.1 Not null

Il valore nullo indica un attributo senza valore. Il suo contrario, ovvero **not null**, indica che il valore nullo non è ammesso come valore dell'attributo e quindi esso deve essere sempre specificato.

Sintassi con esempio:

```
1 Cognome varchar(20) not null
```

13.12.2 Unique

Un vincolo **unique** può essere applicato ad un solo attributo oppure ad un insieme di attributi di una tabella. L'obiettivo è imporre che i valori dell'attributo, o le ennuple di valori sull'insieme di attributi, siano una (super)chiave, ovvero righe differenti della tabella non possano avere gli stessi valori.

Il valore **null** crea un'eccezione in questo caso. Infatti, esso è l'unico valore che ha il permesso di comparire su diverse righe senza violare il vincolo, in quanto si assume che i valori nulli siano tutti diversi tra loro.

La sintassi cambia a seconda dello scopo del suo utilizzo:

1. Definire il vincolo su **un solo** attributo:

```
1 Attributo TipoAttributo unique
```

Per esempio, si vuole rendere unica la matricola di uno studente:

```
1 Matricola character(6) unique
```

2. Definire il vincolo su **un insieme** di attributi:

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

Per esempio, si vuole rendere unico il nome e cognome di una persona, vincolando l'insieme e non i singoli campi:

```
1 Nome      varchar(20) not null,
2 Cognome   varchar(20) not null,
3 unique (Cognome, Nome)
```

Attenzione! C'è differenza tra definire un insieme e, prendendo in considerazione il secondo esempio, vincolare ogni campo:

```
1 Nome      varchar(20) not null unique,
2 Cognome   varchar(20) not null unique
```

Nel caso dell'esempio dell'elenco puntato, si impone che non ci siano due righe che abbiano uguali **sia** il nome, **sia** il cognome. Invece, nel secondo caso qui sopra (più restrittivo), si ha una violazione se nelle righe compaiono più di una volta o lo stesso nome o lo stesso cognome.

13.12.3 Primary key

La chiave primaria può essere specificata tramite il vincolo **primary key** una sola volta per ogni tabella.

Esattamente come il vincolo **unique**, il comando **primary key** può essere definito su un singolo attributo, oppure essere definito elencando più attributi che costituiscono l'identificatore.

Inoltre, gli attributi appartenenti alla chiave primaria **non** possono assumere il valore nullo. Quindi, l'utilizzo di **primary key** implica per tutti gli attributi della chiave primaria una definizione di **not null**.

La sintassi cambia a seconda dello scopo del suo utilizzo:

1. Definire il vincolo su **un solo** attributo:

```
1 Attributo TipoAttributo primary key
```

Per esempio, si vuole rendere come chiave primaria la matricola di uno studente:

```
1 Matricola character(6) primary key
```

2. Definire il vincolo su **un insieme** di attributi:

```
1 primary key ( Attributo { , Attributo } )
```

Per esempio, si vuole rendere come chiave primaria il nome e cognome di una persona:

```
1 Nome      varchar(20),
2 Cognome   varchar(20),
3 primary key (Cognome, Nome)
```

13.13 Vincoli interrelazionali (foreign key)

I vincoli interrelazioni più diffusi e significativi sono i **vincoli di integrità referenziale**. In SQL per la loro definizione viene utilizzato il vincolo **foreign key**, ovvero **chiave esterna**.

Questo vincolo crea un legame tra i valori di un attributo della tabella su cui è definito (**tabella interna**) e i valori di un attributo di un'altra tabella (**tabella esterna**).

Il **vincolo impone** che per ogni riga della tabella interna il valore dell'attributo specificato, se diverso dal valore nullo, sia presente nelle righe della tabella esterna tra i valori del corrispondente attributo.

L'**unico requisito** imposto dalla sintassi è che l'attributo cui si fa riferimento nella tabella esterna sia soggetto ad un vincolo **unique**, quindi sia un identificatore della tabella. Solitamente l'attributo della tabella esterna è la chiave primaria della tabella.

La sintassi cambia a seconda del numero di chiavi esterne:

- Nel caso di **un solo** attributo coinvolto, è possibile utilizzare il costrutto sintattico **references** con il quale si specificano la tabella esterna e l'attributo della tabella esterna al quale l'attributo in questione deve essere legato. Un **esempio** in cui il vincolo su Dipart della tabella Impiegato può assumere solo uno dei valori che le righe della tabella Dipartimento possiedono per l'attributo NomeDip:

```
1 create table Impiegato
2 (
3     Matricola    character(6) primary key,
4     Nome         varchar(20) not null,
5     Cognome      varchar(20) not null,
6     Dipart       varchar(15)
7         references Dipartimento(NomeDip),
8     Ufficio       numeric(3),
9     Stipendio    numeric(9) default 0,
10    unique (Cognome, Nome)
11 )
```

- Nel caso di **un insieme** di attributi coinvolti, è possibili utilizzare il costrutto sintattico **foreign key** in combinazione con **references**, posto al termine della definizione degli attributi.

Se per **esempio** si volesse imporre che gli attributi **Nome** e **Cognome** debbano comparire in una tabella anagrafica, si potrebbe aggiungere il vincolo:

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

La corrispondenza tra gli attributi locali ed esterni avviene in base all'ordine: al primo attributo argomento di **foreign key** corrisponde il primo attributo argomento di **references**, e via via gli altri attributi. In questo caso a **Nome** e **Cognome** di Impiegato corrispondono rispettivamente **Nome** e **Cognome** di Anagrafica.

13.13.1 Politiche di reazione ad una modifica/eliminazione

Solitamente se il sistema rileva una violazione, il comando di aggiornamento viene rifiutato. Al contrario, SQL permette di scegliere altre reazioni da adottare quando viene rilevata una violazione.

Le **violazioni si manifestano** in due occasioni:

1. Inserimento di una nuova riga (operazione sulla tabella interna);
2. Modifica del valore dell'attributo referente (operazione sulla tabella interna).

In questo caso, il sistema le **rifiuta**.

Al contrario, esistono **altre violazioni che sono gestite** a seconda del programmatore:

1. Modifiche del valore dell'attributo riferito (operazione sulla tabella esterna);
2. Cancellazione di righe (operazione sulla tabella esterna).

In altre parole, **le violazioni gestite sono soltanto quelle che si creano operando sulla tabella esterna**. Questo perché la tabella esterna viene rappresentata come la tabella principale (*master*) alle cui violazioni la tabella interna (*slave*) deve adeguarsi.

Per le operazioni di **modifica** sulla tabella esterna, il sistema propone quattro politiche di gestione:

- **cascade**: il **nuovo valore** dell'attributo della tabella esterna viene riportato su tutte le corrispondenti righe della tabella interna;
- **set null**: all'attributo referente viene **assegnato il valore nullo** al posto del valore modificato nella tabella esterna;
- **set default**: all'attributo referente viene **assegnato il valore di default** al posto del valore modificato nella tabella esterna;
- **no action**: l'azione di **modifica non viene consentita** e il sistema non ha quindi bisogno di riparare la violazione.

Invece, per le operazioni di **eliminazione** sulla tabella esterna, il sistema propone quattro politiche di gestione:

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

Si noti che le reazioni alle violazioni possono generare una **reazione a catena** qualora la tabella interna compaia a sua volta come tabella esterna in un altro vincolo di integrità.

La politica di reazione scelta deve essere specificata immediatamente dopo il vincolo di integrità, secondo la seguente **sintassi**:

```
1 on  < delete | update >
2   < cascade | set null | set default | no action >
```

Ovviamente è possibile scegliere un solo valore.

Un **esempio** è il seguente:

```
1 create table Impiegato
2 (
3     Matricola    character(6) primary key,
4     Nome         varchar(20) not null,
5     Cognome      varchar(20) not null,
6     Dipart       varchar(15)
7           references Dipartimento(NomeDip),
8           on delete set null,
9           on update cascade,
10    Ufficio       numeric(3),
11    Stipendio     numeric(9) default 0,
12    primary key(Matricola),
13    unique (Cognome, Nome)
14 )
```

13.14 Vincolo di integrità generico (check)

Per specificare vincoli più complessi, la versione SQL-2 offre la clausola `check` con la seguente sintassi:

```
1 check ( Condizione )
```

Le condizioni ammissibili sono le stesse che possono apparire come argomento della clausola `where` di un'interrogazione SQL. Inoltre, la condizione deve essere **sempre verificata** affinché la base di dati sia corretta. In questo modo è possibile specificare tutti i vincoli intrarelazionali (paragrafo 13.12).

Una dimostrazione della potenza del costrutto consiste nel mostrare come tutti i vincoli predefiniti possano essere descritti con la clausola `check`. Per questo motivo, si riprende la seguente dichiarazione effettuata nel paragrafo dei vincoli interrelazionali:

```
1 create table Impiegato
2 (
3     Matricola    character(6) primary key,
4     Nome         varchar(20) not null,
5     Cognome      varchar(20) not null,
6     Dipart       varchar(15)
7             references Dipartimento(NomeDip),
8     Ufficio       numeric(3),
9     Stipendio     numeric(9) default 0,
10    unique (Cognome, Nome)
11 )
```

E si esegue il confronto utilizzando l'operatore `check`:

```
1 create table Impiegato
2 (
3     Matricola    character(6)
4             check (Matricola is not null and
5                         1 = (select count(*)
6                             from Impiegato I
7                             where Matricola = I.Matricola)),
8     Cognome      character(20) check (Cognome is not null),
9     Nome         character(20) check (Nome is not null and
10                2 > (select count(*)
11                  from Impiegato I
12                  where Nome = I.Nome
13                  and Cognome = I.Cognome)),
14     Dipart       character(15) check (Dipart in
15                 (select NomeDip
16                   from Dipartimento))
17 )
```

È possibile notare diverse cose:

1. I vincoli predefiniti (primo *snippet*) consentono una rappresentazione **molti più compatta e leggibile**. Per esempio, il vincolo di chiave ha bisogno di una rappresentazione abbastanza complicata, che fa uso dell'operatore `count` (contatore/conteggio).
2. **Perdita** della possibilità di associare ai vincoli una **politica di reazione** alle violazioni (paragrafo 13.13.1).
3. Le espressioni dei **vincoli mediante costrutti predefiniti** (primo *snippet*) viene gestita in modo migliore dal sistema, il quale le riconosce immediatamente e riesce a gestirle in modo **più efficiente**.

Un utilizzo più sensato della clausola `check`, è possibile esprimere grazie alla seguente richiesta (e.g.): si richiede che un impiegato abbia un manager del proprio dipartimento, a meno che la matricola non inizi con la cifra 1. Per soddisfare questa richiesta, basta estendere la dichiarazione della tabella `Impiegato` con le seguenti dichiarazioni:

```
1 create table Impiegato
2 (
3     Matricola    character(6) primary key,
4     Nome         varchar(20) not null,
5     Cognome      varchar(20) not null,
6     Dipart       varchar(15)
7             references Dipartimento(NomeDip),
8     Ufficio       numeric(3),
9     Stipendio     numeric(9) default 0,
10    Superiore    character(6),
11    check        (Matricola like '1%' or
12                  Dipart = (select Dipart
13                           from Impiegato I
14                         where I.Matricola = Superiore))
15    unique        (Cognome, Nome)
16 )
```

13.15 Modifica degli schemi

13.15.1 Modifica dei domini e schemi (`alter`)

Il comando `alter` consente di modificare domini e schemi di tabelle. La sua sintassi è la seguente:

```
1 alter domain NomeDominio < set default ValoreDefault |  
2     drop default |  
3     add constraint DefVincolo |  
4     drop constraint NomeVincolo >  
5 alter table NomeTabella <  
6     alter column NomeAttributo < set default NuovoDefault |  
7         drop default > |  
8     add constraint DefVincolo |  
9     drop constraint NomeVincolo |  
10    add column DefAttributo |  
11    drop column NomeAttributo >
```

- Con `alter domain` è possibile aggiungere e rimuovere vincoli sul dominio, e modificare i valori di default associati ai domini;
- Con `alter table` è possibile aggiungere e rimuovere vincoli sul dominio, e modificare i valori di default associati agli attributi.

Quando viene definito un nuovo vincolo, questo deve essere soddisfatto dai dati già presenti. Nel caso in cui l'istanza contiene violazioni per il nuovo vincolo, l'inserimento viene rifiutato.

Per **esempio**, in questo caso viene esteso lo schema della tabella Dipartimento con un attributo `NroUff` che consente di rappresentare il numero di uffici di cui il dipartimento è dotato:

```
1 alter table Dipartimento add column NroUff numeric(4)
```

13.15.2 Rimuovere schemi, domini, viste, asserzioni (`drop`)

Il comando `drop` consente di rimuovere dei componenti quali schemi, domini, tabelle, viste o asserzioni⁷. La sua sintassi è la seguente:

```
1 drop < schema | domain | table | view | assertion > NomeElemento  
2   [ restrict | cascade ]
```

- Con `restrict` viene specificato che il comando di rimozione **non** deve essere eseguito nel caso in cui ci siano oggetti **non vuoti**. Quindi:
 - **Schema** non viene rimosso contiene tabelle o altri oggetti;
 - **Dominio** non viene rimosso se appare in qualche definizione di tabella;
 - **Tabella** non viene rimossa se possiede delle righe o se è presente in qualche definizione di tabella o vista;
 - **Vista** non viene rimossa se è utilizzata nella definizione di tabelle o viste.

Il valore `restrict` è l'**opzione di default**.

- Con `cascade` tutti gli oggetti vengono rimossi:
 - **Schema** non vuoto, tutti gli oggetti che fanno parte dello schema vengono eliminati;
 - **Dominio** che compare nella definizione di qualche attributo, la sua rimozione fa sì che il nome del dominio venga rimosso, ma gli attributi che sono stati definiti utilizzando quel dominio rimangono associati al medesimo dominio elementare.
Per **esempio**, eliminando il dominio `StringaLunga`, definito come `varchar(100)`, tramite il comando:

```
1 drop domain StringaLunga cascade
```

Tutti gli attributi definiti su quel dominio assumeranno direttamente il dominio `varchar(100)`;

- **Tabella**, tutte le righe vengono perse. Nel caso in cui la tabella compariva in qualche definizione di tabella o vista, anch'esse vengono rimosse.
- **Vista** che compare nella definizione di altre tabelle o viste, anche queste tabelle e viste vengono rimosse.

Data la potenza di `cascade` e degli effetti disastrosi che potrebbe causare per cascata, è necessario prestare estrema cautela nell'uso di questa opzione, in quanto può capitare che, a causa di qualche dipendenza sfuggita all'analisi, il comando abbia un effetto molto diverso da quello voluto.

⁷Le **asserzioni** sono dei vincoli che non sono associati ad alcuna tabella in particolare.

13.16 Modifica dei dati

13.16.1 Inserimento (`insert into`)

Il comando di inserimento `insert into` presenta un'unica sintassi con un possibile doppio utilizzo:

```
1 insert into NomeTabella [ ListaAttributi ]
2   < values( ListaValori ) |
3     SelectSQL >
```

1. È possibile inserire **singole righe** all'interno delle tabelle. L'argomento `values` rappresenta esplicitamente i valori degli attributi della singola riga.

Per **esempio**:

```
1 insert into Dipartimento(NomeDip, Citta)
2   values('Produzione', 'Torino')
```

Questa forma viene usata tipicamente all'interno dei programmi per riempire una tabella con i dati forniti direttamente dagli utenti.

2. È possibile inserire **insiemi di righe**, estratti dal contenuto della base di dati.

Per **esempio**, con la seguente query viene inserito nella tabella `ProdottiMilanesi` il risultato della selezione della relazione `Prodotto` di tutte le righe aventi 'Milano' come valore dell'attributo `LuogoProd`:

```
1 insert into ProdottiMilanesi
2   (select Codice, Descrizione
3    from Prodotto
4    where LuogoProd = 'Milano')
```

Questa forma consente di inserire i dati in una tabella a partire da altre informazioni presenti nella base di dati.

Una regola che vale indipendentemente dall'utilizzo scelto: se in un inserimento **non vengono specificati i valori** di tutti gli attributi della tabella, agli attributi mancanti viene assegnato il valore di default, o in assenza di questo il valore nullo.

Infine, la corrispondenza tra gli attributi della tabella e i valori da inserire è data dall'ordine in cui compaiono i termini nella definizione della tabella.

13.16.2 Cancellazione (delete, drop)

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

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

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

Inoltre, si ricorda che qualora esista un **vincolo di integrità referenziale** con politica di **cascade** in cui la tabella viene referenziata, allora la cancellazione di righe dalla tabella può comportare la cancellazione di righe appartenente ad altre tabelle (reazione a catena).

Un **esempio** di eliminazione di una riga da Dipartimento avente come nome 'Produzione':

```
1 delete from Dipartimento  
2      where NomeDip = 'Produzione'
```

Un altro **esempio** in cui compare al suo interno anche interrogazioni nidificate che fanno riferimento ad altre tabelle:

```
1 delete from Dipartimento  
2      where Nome not in (select Dipart  
3                           from Impiegato)
```

Eliminazione dei dipartimenti senza impiegati.

Infine, si noti la **differenza** tra il comando **delete** appena visto e il comando **drop** (paragrafo 13.15.2). Infatti, un comando:

```
1 delete from Dipartimento
```

Elimina tutte le righe della tabella Dipartimento, eventualmente eliminando anche tutte le righe dalle tabella che sono legate da vincolo di integrità referenziale con la tabella, se per il vincolo è specificata la politica **cascade**.

Invece, con il comando:

```
1 drop table Dipartimento cascade
```

Si ha lo stesso effetto del **delete**, ma in più anche lo **schema** della base di dati viene **modificato**, eliminando dallo schema non solo la tabella Dipartimento, ma anche tutte le viste e tabelle che nella loro definizione fanno riferimento ad essa.

13.16.3 Modifica (`update`)

Il comando `update` presenta una sintassi leggermente complicata:

```
1 update NomeTabella
2   set Attributo = < Espressione | SelectSQL | null | default >
3   {, Attributo = < Espressione | SelectSQL | null | default > }
4 [ where Condizione ]
```

Con `update` è possibile aggiornare uno o più attributi delle righe di `NomeTabella` che soddisfano l'eventuale `Condizione`. Se il comando non presenta la clausola `where`, come al solito viene supposto che la condizione sia soddisfatta e viene eseguita la modifica su tutte le righe.

Il nuovo valore può essere:

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

Un **esempio** di modifica di una singola riga aggiornando lo stipendio del dipendente con matricola M2047:

```
1 update Dipendente
2   set Stipendio = StipendioBase + 5
3   where Matricola = 'M2047'
```

Un **esempio** di modifica di un insieme di righe aumentando del 10% lo stipendio di tutti gli impiegati che lavorano in Amministrazione:

```
1 update Impiegato
2   set Stipendio = Stipendio * 1.1
3   where Dipart = 'Amministrazione'
```

13.17 Interrogazioni semplici

13.17.1 Select

La clausola **select** specifica gli elementi dello schema della tabella risultato. Come argomento può comparire anche il carattere speciale * (asterisco), il quale rappresenta la **selezione di tutti gli attributi delle tabelle elencate nella clausola from**.

La sua sintassi:

```
1 select AttrEspr [[as] Alias] {, AttrEspr[[as] Alias]}
2 from Tabella [[as] Alias] {, Tabella[[as] Alias]}
3 [where Condizione]
```

- AttrEspr sono le colonne che si ottengono dalla valutazione delle espressioni che appaiono nella clausola **select**;
- Alias è la ridefinizione che può subire una colonna;
- from è la tabella su cui cercare la selezione;
- where è la condizione che filtra i risultati.

Per **esempio**, data la base di dati composta da due tabelle:

- Impiegato(Nome, Cognome, Dipart, Ufficio, Stipendio, Città)
- Dipartimento(Nome, Indirizzo, Città)

(dichiarazione a pag.159) si eseguono le seguenti interrogazioni:

1. Estrazione di tutte le informazioni relative agli impiegati di cognome “Rossi”:

```
1 select *
2 from Impiegato
3 where Cognome = 'Rossi'
```

Il risultato:

Nome	Cognome	Dipart	Ufficio	Stipendio	Città
Mario	Rossi	Amministrazione	10	45	Milano
Carlo	Rossi	Direzione	14	80	Milano

2. Estrazione dello stipendio mensile dell’impiegato che ha cognome “Bianchi”:

```
1 -- Diviso 12 perche' il campo Stipendio e' annuale
2 select Stipendio/12 as StipendioMensile
3 from Impiegato
4 where Cognome = 'Bianchi'
```

Il risultato è un unico valore (3,00) avente come nome della colonna StipendioMensile.

Impiegato

Nome	Cognome	Dipart	Ufficio	Stipendio	Città
Mario	Rossi	Amministrazione	10	45	Milano
Carlo	Bianchi	Produzione	20	36	Torino
Giovanni	Verdi	Amministrazione	20	40	Roma
Franco	Neri	Distribuzione	16	45	Napoli
Carlo	Rossi	Direzione	14	80	Milano
Lorenzo	Gialli	Direzione	7	73	Genova
Paola	Rosati	Amministrazione	75	40	Venezia
Marco	Franco	Produzione	20	46	Roma

Dipartimento

Nome	Indirizzo	Città
Amministrazione	Via Tito Livio, 27	Milano
Produzione	P.le Lavater, 3	Torino
Distribuzione	Via Segre, 9	Roma
Direzione	Via Tito Livio, 27	Milano
Ricerca	Via Venosa, 6	Milano

Figura 44: DBMS d'esempio.

13.17.2 From

Quando viene considerata un'interrogazione che coinvolge righe appartenenti a più di una tabella, viene posto come argomento della clausola **from** l'**insieme di tabelle alle quali si vuole accedere**. Sul prodotto cartesiano delle tabelle elencate verranno applicate le condizioni contenute nella clausola **where**. Sintassi identica al paragrafo 13.17.1.

Per **esempio**, data la base dati a pagina 159, si eseguono le seguenti interrogazioni:

1. Estrazione dei nomi degli impiegati e le città in cui lavorano:

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

Il risultato:

Impiegato.Nome	Impiegato.Cognome	Dipartimento.Città
Mario	Rossi	Milano
Carlo	Bianchi	Torino
Giovanni	Verdi	Milano
Franco	Neri	Roma
Carlo	Rossi	Milano
Lorenzo	Gialli	Milano
Paola	Rosati	Milano
Marco	Franco	Torino

2. Data la precedente ambiguità tra Nome e Città, si riformula la query in modo corretto (il risultato rimane invariato, escluso il nome delle colonne che si abbreviano):

```
1 select I.Nome, Cognome, D.Città
2 from Impiegato as I, Dipartimento as D
3 where Dipart = D.Nome
```

13.17.3 Where e i suoi operatori (like, similar to, between, in, is null)

La clausola `where` ammette come argomento un'espressione booleana costruita combinando predicati semplici con gli operatori:

- *and*
- *or*
- *not*

Ogni predicato semplice utilizza gli operatori:

- = uguale
- <> diverso
- < minore
- > maggiore
- ≤ minore-uguale
- ≥ maggiore-uguale

Per **esempio**, data la base dati a pagina 159, si eseguono le seguenti interrogazioni:

1. Estrazione del nome e cognome degli impiegati che lavorano nell'ufficio 20 del dipartimento Amministrazione:

```
1 select Nome, Cognome
2 from Impiegato
3 where Ufficio = 20 and
4       Dipart = 'Amministrazione'
```

Il risultato è una riga con Nome Giovanni e Cognome Verdi.

2. Estrazione dei nomi e cognomi degli impiegati che lavorano nel dipartimento Amministrazione o nel dipartimento Produzione:

```
1 select Nome, Cognome
2 from Impiegato
3 where Dipart = 'Amministrazione' or
4       Dipart = 'Produzione'
```

Il risultato:

Nome	Cognome
Mario	Rossi
Carlo	Bianchi
Giovanni	Verdi
Paola	Rosati
Marco	Franco

3. Estrazione dei nomi propri degli impiegati di cognome “Rossi” che lavorano nei dipartimenti Amministrazione o Produzione:

```

1 select Nome,
2 from Impiegato
3 where Cognome = 'Rossi' and
4       (Dipart = 'Amministrazione' or
5        Dipart = 'Produzione')

```

Il risultato è una riga con Nome Mario.

La clausola `where` implementa **due operatori** abbastanza simili: `like` e `similar to`. L’operatore `like` viene utilizzato per confronto di stringhe e consente anche di effettuare confronti con stringhe in cui compaiono i caratteri speciali:

- `_` (trattino sottolineato), utilizzato per rappresentare un carattere arbitrario
- `%` (percentuale), utilizzato per rappresentare un numero arbitrario (anche nullo) di caratteri arbitrari

Per **esempio**, un confronto file ‘`ab%ba_`’ è soddisfatto quando qualsiasi stringa di caratteri che inizia con `ab` e che ha la coppia di caratteri `ba` prima dell’ultima posizione. Una *query* d’**esempio** sempre considerando la base dati a pagina 159:

```

1 select *
2 from Impiegato
3 where Cognome like '_o%o'

```

Il risultato:

Nome	Cognome	Dipart	Ufficio	Stipendio	Città
Mario	Rossi	Amministrazione	10	45	Milano
Carlo	Rossi	Direzione	14	80	Milano
Paola	Rosati	Amministrazione	75	40	Venezia

L’operatore `similar to` è come la clausola `where` ma ha una rosa di operatori molto più ampia. Infatti, essa accetta come *pattern* un sottoinsieme delle espressioni regolari POSIX. Alcune delle espressioni regolari più utilizzate:

- `_` (trattino sottolineato), un carattere qualsiasi
- `%` (percentuale), nessuno o più caratteri qualsiasi
- `*` (asterisco), ripetizione del precedente match 0 o più volte
- `+` (segno addizione), ripetizione del precedente *match* una o più volte
- `{n,m}`, ripetizione del precedente *match* almeno *n* volte e non oltre *m* volte
- `[...]`, al posto dei punti si inserisce un elenco di caratteri ammissibili (*white list*)

Si lascia il [link alla documentazione](#) ufficiale per chiarire eventuali dubbi.

Nella clausola `where` può apparire anche l'**operatore between** per testare l'appartenenza (o no) di un valore ad un intervallo (estremi compresi!).

La sintassi è la seguente:

```
1 where Attr [not] between ValoreIniziale and ValoreFinale
```

Un **esempio** sempre considerando la base dati a pagina 159:

```
1 select *
2 from Impiegato
3 where Stipendio between 45 and 50
```

Il risultato:

Nome	Cognome	Dipart	Ufficio	Stipendio	Città
Mario	Rossi	Amministrazione	10	45	Milano
Franco	Neri	Distribuzione	16	45	Napoli
Marco	Franco	Produzione	20	46	Roma

Nella clausola `where` può apparire anche l'**operatore in** per testare l'appartenenza (o no) di un valore ad un insieme.

La sintassi è la seguente:

```
1 where Attr [not] in (valore [, ...])
```

Un **esempio** sempre considerando la base dati a pagina 159:

```
1 select *
2 from Impiegato
3 where Stipendio in(45, 46)
```

Il risultato:

Nome	Cognome	Dipart	Ufficio	Stipendio	Città
Mario	Rossi	Amministrazione	10	45	Milano
Franco	Neri	Distribuzione	16	45	Napoli
Marco	Franco	Produzione	20	46	Roma

Infine, per selezionare i termini con valori nulli SQL fornisce il predicato **is null**, la cui sintassi è semplicemente:

```
1 where Attr is [not] null
```

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

Si consideri un predicato di confronto tra un attributo e un valore costante:

```
Stipendio > 40
```

Questo predicato sarà vero per le righe in cui l'attributo **Stipendio** è superiore a 40.

13.18 Ordinamento (order by)

SQL consente di specificare un ordinamento delle righe del risultato di un'interrogazione tramite la clausola **order by**, con la quale si chiude l'interrogazione. La sintassi:

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

Si specificano gli attributi che devono essere usati per l'ordinamento. Le righe vengono **ordinate in base al primo attributo nell'elenco**. Nel caso in cui due righe hanno lo stesso valore per il primo attributo, si considerano i valori degli attributi successivi in sequenza. L'ordine può essere ascendente (**asc**) o discendente (**desc**) e di default è **asc**.

Un **esempio** sempre considerando la base dati a pagina 159:

```
1 select *
2 from Impiegato
3 order by Dipart desc, Nome
```

Il risultato:

Impiegato

Nome	Cognome	Dipart	Ufficio	Stipendio	Città
Carlo	Bianchi	Produzione	20	36	Torino
Marco	Franco	Produzione	20	46	Roma
Franco	Neri	Distribuzione	16	45	Napoli
Carlo	Rossi	Direzione	14	80	Milano
Lorenzo	Gialli	Direzione	7	73	Genova
Giovanni	Verdi	Amministrazione	20	40	Roma
Mario	Rossi	Amministrazione	10	45	Milano
Paola	Rosati	Amministrazione	75	40	Venezia

13.19 Operatori aggregati (`count`, `sum`, `max`, `min`, `avg`)

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

L'operatore `count` ha la seguente sintassi:

```
1 count ( < * | [distinct | all] ListaAttributi > )
```

- `*` restituisce il numero di righe;
- `distinct` restituisce il numero di diversi valori degli attributi in `ListaAttributi`;
- `all` (default) restituisce il numero di righe che possiedono valori diversi dal valore nullo per gli attributi in `ListaAttributi`.

Per **esempio**, si eseguono le seguenti *query* (sempre considerando la base dati a pagina 159):

- Si vuol estrarre il numero di diversi valori dell'attributo `Stipendio` fra tutte le righe di `Impiegato`:

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

Il risultato è 6 perché i valori distinti sono 45, 36, 40, 80, 73, 46.

- Si vuole estrarre il numero di righe che possiedono un valore non nullo per l'attributo `Nome`:

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

Il risultato è 8 perché tutte le tuple hanno un valore nel campo `Nome`.

Gli operatori `sum` e `avg` ammettono come argomento solo espressioni che rappresentano **valori numerici o intervalli di tempo**. Invece, gli operatori `max` e `min` richiedono soltamente che sull'**espressione sia definito un ordinamento**, per cui si possono applicare anche su stringhe di caratteri o su istanti di tempo. La sintassi:

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

- `sum` restituisce la somma dei valori posseduti dall'espressione;
- `max` restituisce il valore massimo;
- `min` restituisce il valore minimo;
- `avg` restituisce la media dei valori, ovvero la divisione tra `sum` e `count`.
- `distinct` elimina i duplicati (non ha effetto su `max` e `min`);
- `all` trascura solo i valori nulli (non ha effetto su `max` e `min`).

Per **esempio**, si eseguono le seguenti *query* (sempre considerando la base dati a pagina 159):

- Si vuole estrarre la somma degli stipendi del dipartimento Amministrazione:

```
1 select sum(Stipendo)
2 from Impiegato
3 where Dipart = 'Amministrazione'
```

Il risultato è 125.

- Si vuole estrarre lo stipendio minimo, massimo e medio fra quelli di tutti gli impiegati:

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

Il risultato è 36 per il valore minimo, 80 per il valore massimo e 50,625 per la media.

- Si vuole estrarre il massimo stipendio tra quelli degli impiegati che lavorano in un dipartimento con sede a Milano:

```
1 select max(Stipendio)
2 from Impiegato, Dipartimento D
3 where Dipart = D.Nome and
4      D.Città = 'Milano'
```

Il risultato è 80.

- Questa *query* **non è corretta**:

```
1 select Cognome, Nome, max(Stipendio)
2 from Impiegato, Dipartimento D
3 where Dipart = D.Nome and
4      D.Città = 'Milano'
```

Questa interrogazione non può essere utilizzata poiché gli operatori aggregati (in questo caso `max`) non rappresentano un meccanismo di selezione, ma solo delle funzioni che restituiscono un valore quando sono applicate ad un insieme. Quindi, **SQL non ammette che nella stessa clausola `select` compaiano funzioni aggregati ed espressioni al livello di riga, a meno che non si faccia utilizzo della clausola `group by`.**

13.20 Join interni ed esterni (inner/right/left/full/cross join)

La sintassi del join è la seguente:

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

Il parametro **TipoJoin** rappresenta qual è il tipo di join da utilizzare che può essere:

- **Valore di default:** **inner join** è il **tradizionale theta-join** dell'algebra relazionale (paragrafo 5.3.3);
- **cross join** è il **prodotto cartesiano** di due o più tabelle;
- **right join** fornisce come risultato il **join interno esteso con le righe della tabella che compare a destra per le quali non esiste una corrispondente riga nella tabella di sinistra**;
- **left join** fornisce come risultato il **join interno esteso con le righe della tabella che compare a sinistra per le quali non esiste una corrispondente riga nella tabella di destra**;
- **full join** restituisce il **join interno esteso con le righe escluse di entrambe le tabelle**.

È possibile aggiungere la parola **natural** davanti ogni join così da specificare l'utilizzo del join naturale, ovvero utilizzare un join con una condizione implicita di uguaglianza su tutti gli attributi caratterizzati dallo stesso nome.

Si presentano alcuni **esempi**:

- Tenendo in considerazione la base dati a pagina 13.17.1, si vogliono estrarre i nomi degli impiegati e le città in cui lavorano:

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

Il risultato:

I.Nome	Cognome	D.Città
Mario	Rossi	Milano
Carlo	Bianchi	Torino
Giovanni	Verdi	Milano
Franco	Neri	Roma
Carlo	Rossi	Milano
Lorenzo	Gialli	Milano
Paola	Rosati	Milano
Marco	Franco	Torino

- Si consideri la seguente base dati:

Guidatore

Nome	Cognome	NroPatente
Mario	Rossi	VR 2030020Y
Carlo	Bianchi	PZ 1012436B
Marco	Neri	AP 4544442R

Automobile

Targa	Marca	Modello	NroPatente
KB 574 WW	Fiat	Punto	VR 2030020Y
GA 652 FF	Fiat	Punto	VR 2030020Y
BJ 747 XX	Lancia	Ypsilon	PZ 1012436B
ZB 421 JJ	Fiat	Uno	MI 2020030U

Figura 45: Base dati d'esempio.

- Si vuole estrarre i guidatori con le automobili loro associate, mantenendo nel risultato anche i guidatori senza automobile:

```

1 select Nome , Cognome , G.NroPatente ,
2      Targa , Marca , Modello
3 from Guidatore G left join Automobile A
4   on (G.NroPatente = A.NroPatente)

```

Si noti che l'ultima riga del risultato rappresenta un guidatore cui non risulta associata nessuna automobile:

Nome	Cognome	G.NroPatente	Targa	Marca	Modello
Mario	Rossi	VR 2030020Y	KB 574 WW	Fiat	Punto
Mario	Rossi	VR 2030020Y	GA 652 FF	Fiat	Panda
Carlo	Bianchi	PZ 1012436B	BJ 747 XX	Lancia	Ypsilon
Marco	Neri	AP 4544442R	NULL	NULL	NULL

- Si vuole estrarre tutti i guidatori e tutte le auto, mostrando tutte le relazioni esistenti tra di essi:

```

1 select Nome, Cognome, G.NroPatente,
2      Targa, Marca, Modello
3 from Guidatore G full join Automobile A
4   on (G.NroPatente = A.NroPatente)

```

L’interrogazione produce come risultato la seguente tabella. Si noti che l’ultimo elemento della tabella descrive un’automobile per la quale non esiste un corrispondente elemento in Guidatore:

Nome	Cognome	G.NroPatente	Targa	Marca	Modello
Mario	Rossi	VR 2030020Y	KB 574 WW	Fiat	Punto
Mario	Rossi	VR 2030020Y	GA 652 FF	Fiat	Panda
Carlo	Bianchi	PZ 1012436B	BJ 747 XX	Lancia	Ypsilon
Marco	Neri	AP 4544442R	NULL	NULL	NULL
NULL	NULL	NULL	ZB 421 JJ	Fiat	Uno

- Si estraggono tutti i guidatori e tutte le auto, mostrando tutte le relazioni esistenti tra di essi, ma questa volta usando il join naturale:

```

1 select Nome, Cognome, G.NroPatente,
2      Targa, Marca, Modello
3 from Guidatore G natural full join Automobile

```

Il risultato è identico al punto precedente (tabella soprastante). Analizzando il join naturale, esso **non è normalmente consigliabile** poiché un’interrogazione che lo utilizza può introdurre dei rischi nelle applicazioni, in quanto il suo comportamento può mutare profondamente al variare dello schema delle tabelle.

13.21 Raggruppamento (group by)

Spesso nasce l'esigenza di **applicare l'operatore aggregato separatamente a sottoinsiemi di righe**. Per poter utilizzare in questo modo l'operatore aggregato, SQL mette a disposizione la clausola **group by**, la quale consente di specificare come dividere le tabelle in sottoinsiemi.

La sintassi e il suo funzionamento vengono spiegate con l'introduzione di un esempio: si vuole estrarre la somma degli stipendi di tutti gli impiegati dello stesso dipartimento (si consideri la base dati a pagina 159):

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

I passi eseguiti per ottenere il risultato sono:

1. L'interrogazione viene eseguita come se la clausola **group by** non esistesse. Quindi, viene eseguita una **selezione degli attributi** che appaiono **come argomento della clausola group by** o che **compaiono all'interno dell'espressione argomento dell'operatore aggregato**.

Nell'esempio è come se si considerasse:

```
1 select Dipart, Stipendio
2 from Impiegato
```

Dipart	Stipendio
Amministrazione	45
Produzione	36
Amministrazione	40
Distribuzione	45
Direzione	80
Direzione	73
Amministrazione	40
Produzione	46

2. La tabella ottenuta, precisamente le righe (*records*), viene **divisa in insiemi caratterizzati dallo stesso valore degli attributi che compaiono come argomento della clausola group by**. Nell'esempio vengono raggruppate in base al valore dell'attributo Dipart, il risultato è il seguente:

Dipart	Stipendio
Amministrazione	45
Amministrazione	40
Amministrazione	40
Produzione	36
Produzione	46
Distribuzione	45
Direzione	80
Direzione	73

3. Viene **applicato l'operatore aggregato separatamente su ogni sottinsieme**. Il risultato è una tabella con righe che contengono l'esito della valutazione dell'operatore aggregato affiancato al valore dell'attributo che è stato usato per l'aggregazione.
- Nell'esempio il risultato finale:

Dipart	sum(Stipendio)
Amministrazione	125
Produzione	82
Distribuzione	45
Direzione	153

ATTENZIONE: un'interrogazione può utilizzare la clausola **group by** se compare come **argomento della select solamente un sottoinsieme degli attributi usati nella clausola group by**. Infatti, per questi attributi, ciascuna tupla del sottoinsieme sarà caratterizzata dallo stesso valore. Un esempio che non rispetta tale regola, viene presentato nella prossima pagina.

Alcuni **esempi** particolari:

- Questo caso mostra i **problemi** che possono essere introdotti da interrogazioni che presentano nella clausola `select` attributi che non appaiono nella clausola `group by`:

```
1 select Ufficio  
2 from Impiegato  
3 group by Dipart
```

È un'**interrogazione scorretta!** Infatti, ad ogni valore dell'attributo `Dipart` corrisponderanno diversi valori dell'attributo `Ufficio`.

Al contrario, dopo l'esecuzione del raggruppamento, ogni sottoinsieme di righe deve corrispondere ad una sola riga nella tabella risultato dell'interrogazione.

- Un'altra **interrogazione vietata** è la seguente:

```
1 select Dipart, count(*), D.Citta  
2 from Impiegato I join Dipartimento D  
3   on (I.Dipart = D.Nome)  
4 group by Dipart
```

Questa interrogazione **dovrebbe** restituire i dipartimenti, il numero di impiegati di ciascun dipartimento e la città in cui il dipartimento ha sede. Dato che `Nome` è chiave di `Dipartimento`, ad ogni valore di `Dipart` corrisponde un preciso valore di `Citta`.

L'interrogazione può essere riscritta correttamente nel seguente modo:

```
1 select Dipart, count(*), D.Citta  
2 from Impiegato I join Dipartimento D  
3   on (I.Dipart = D.Nome)  
4 group by Dipart, D.Citta
```

13.21.1 Predicati sui gruppi (having)

A seconda delle necessità, si possono presentare due casistiche:

1. **Problema:** considerare solo sottoinsiemi che soddisfano condizioni verificabili al livello delle singole righe.

Soluzione: porre i predicati come argomento nella clausola where.

2. **Problema:** considerare solo sottoinsiemi che soddisfano condizioni di tipi aggregato.

Soluzione: costrutto having, il quale **descrive le condizioni che si devono applicare al termine dell'esecuzione di un'interrogazione che fa uso della clausola group by**

Quindi, il costrutto having consente di filtrare i risultati della clausola group by.

È inoltre possibile utilizzare il costrutto having anche senza group by. In tal caso, l'intero insieme di righe è trattato come un unico raggruppamento. Solitamente, questo metodo è poco utilizzato poiché in caso di condizione falsa, il risultato sarà vuoto.

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

Per **esempio**:

- Si vuole estrarre i dipartimenti che spendono più di 100 in stipendi:

```
1 select Dipart, sum(Stipendio) as SommaStipendi
2 from Impiegato
3 group by Dipart
4 having sum(Stipendio) > 100
```

Vengono raggruppate le righe in base al valore dell'attributo Dipart; viene valutato il predicato argomento della clausola having, la quale seleziona i dipartimenti per cui la somma degli stipendi, per tutti gli elementi del sottoinsieme, è superiore a 100. Il risultato:

Dipart	SommaStipendi
Amministrazione	125
Direzione	153

- Si vuole estrarre i dipartimenti per cui la media degli stipendi degli impiegati che lavorano nell'ufficio 20 è superiore a 25:

```
1 select Dipart
2 from Impiegato
3 where Ufficio = 20
4 group by Dipart
5 having sum(Stipendio) > 25
```

13.22 Interrogazioni nidificate (any, all, in, not in, some)

Il linguaggio SQL consente di scrivere interrogazioni che presentano al loro interno altre interrogazioni. Si parla in questo caso di **interrogazioni nidificate** e possono avvenire sia nella clausola `select`, sia nella clausola `from`, sia nella clausola `where`. In questo paragrafo verranno viste nella clausola `where` poiché è l'uso più comune. Mentre nel paragrafo 13.22.4 è possibile vederle nelle clausole `select` e `from`.

13.22.1 Operatori any/some e all

SQL mette a disposizione i comandi `all` e `any` (o il suo sinonimo `some`), con l'obiettivo di **estendere** i classici **operatori di confronto**: `=`, `<>`, `<`, `>`, `<=`, `>=`.

L'operatore `any` specifica che la riga **soddisfa la condizione** se risulta vero il **confronto** (con l'operatore specificato) tra il **valore dell'attributo per la riga** e **almeno uno degli elementi** restituiti dall'interrogazione. Il sinonimo `some` ha le stesse caratteristiche di `any`.

L'operatore `all` specifica che la riga **soddisfa la condizione solo se tutti gli elementi** restituiti dall'interrogazione **nidificata** rendono vero il **confronto**.

La sintassi richiede la **compatibilità di dominio** tra l'attributo restituito dall'interrogazione **nidificata** e l'attributo con cui avviene il **confronto**. Quindi non possono avvenire, per esempio, confronto tra una stringa e un intero.

Alcune interrogazioni di **esempio**:

- Si vuole estrarre gli impiegati che lavorano in dipartimenti situati a Firenze:

```
1 select *
2 from Impiegato
3 where Dipart = any (select Nome
4                      from Dipartimento
5                      where Citta = 'Firenze')
```

L'interrogazione seleziona le righe di `Impiegato` per cui il valore dell'attributo `Dipart` è uguale ad almeno uno dei valori dell'attributo `Nome` delle righe di `Dipartimento`.

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

- Si vogliono trovare gli impiegati che hanno lo stesso nome di un impiegato nel dipartimento Produzione. L’interrogazione può essere espressa in diversi modi:

- La più compatta e con l’utilizzo di variabili:

```

1 select I1.Nome
2 from Impiegato I1, Impiegato I2
3 where I1.Nome = I2.Nome and
4   I2.Dipart = 'Produzione'
```

- Utilizzando un’interrogazione nidificata:

```

1 select Nome
2 from Impiegato
3 where Nome = any (select Nome
4                     from Impiegato
5                     where Dipart = 'Produzione')
```

- Si vogliono estrarre i dipartimenti in cui non lavorano persone di cognome “Rossi”:

```

1 select Nome
2 from Dipartimento
3 where Nome <> all (select Dipart
4                      from Impiegato
5                      where Cognome = 'Rossi')
```

L’interrogazione nidificata seleziona i valori di Dipart di tutte le righe in cui il cognome vale “Rossi”. La condizione viene soddisfatta da quelle righe di Dipartimento per cui il valore dell’attributo Nome non fa parte dei nomi prodotti dall’interrogazione nidificata.

Questa interrogazione **non** poteva essere **espressa mediante un join**.

13.22.2 Operatori **in** e **not in**

Per rappresentare il **controllo di appartenenza e di esclusione rispetto ad un insieme**, SQL mette a disposizione due appositi operatori, **in** e **not in**, i quali risultano del tutto identici agli operatori che sono stati visti negli esempi del paragrafo precedente (= **any**, <**>** **all**).

Data la semplicità di questi operatori, gli **esempi** sono consultabili in combinazione con altri argomenti al paragrafo 13.22.3.

13.22.3 Interrogazioni nidificate complesse (`exists`), passaggio di binding e scope

Talvolta l'interrogazione **nidificata** fa riferimento al contesto dell'**interrogazione che la racchiude**; tipicamente ciò accade tramite una variabile definita nell'ambito della *query* più esterna e usata nell'ambito della *query* più interna (si parla di un **passaggio di binding** da un contesto all'altro). La presenza di questo meccanismo arricchisce il potere espressivo di SQL e consente di comprendere meglio l'operatore che verrà introdotto tra poco.

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

- Si vogliono estrarre gli impiegati che afferiscono al dipartimento Produzione o a un dipartimento che risiede nella stessa città del dipartimento Produzione:

```
1 select *
2 from Impiegato
3 where Dipart in (select Nome
4                   from Dipartimento D1
5                   where Nome = 'Produzione') or
6       Dipart in (select Nome
7                   from Dipartimento D2
8                   where D1.Citta = D2.Citta) -- D1 e' sbagliato
```

La *query* non rispetta la sintassi SQL perché utilizza la variabile **D1** dove non è visibile.

L'operatore logico `exists` ammette come **parametro** un'interrogazione **nidificata** e restituisce il valore **vero solo** se l'interrogazione fornisce un **risultato non vuoto** (quantificatore esistenziale della logica \exists). L'operatore può essere utilizzato in modo significativo quando vi è un *passaggio di binding* tra l'interrogazione esterna e quella nidificata.

Per **esempio**, si consideri una relazione che descrive dati anagrafici, avente il seguente schema:

Persona (CodFiscale, Nome, Cognome, Città)

- Si vuol estrarre le persone che hanno degli omonimi, ovvero persone con lo stesso nome e cognome, ma diverso codice fiscale:

– Utilizzando una *query* nidificata:

```

1 select *
2 from Persona P
3 where exists (select *
4                 from Persona P1
5                 where P1.Nome = P.Nome and
6                       P1.Cognome = P.Cognome and
7                       P1.CodFiscale <> P.CodFiscale)

```

Si osservi che in questo caso **non risulta possibile eseguire l'interrogazione nidificata prima di valutare l'interrogazione più esterna**, in quanto senza aver associato un valore alla variabile P l'interrogazione nidificata non risulta completamente definita.

Prima deve essere valutata l'interrogazione esterna; per ogni singola riga esaminata nell'ambito dell'interrogazione esterna si deve valutare l'interrogazione nidificata. In questo modo, prima di tutto verranno considerate una a una le righe associate alla variabile P; per ciascuna di esse sarà poi eseguita l'interrogazione nidificata che restituirà o meno l'insieme vuoto a seconda che vi siano o meno degli omonimi della persona.

– Utilizzando un join tra due diversi istanze della tabella Persona:

```

1 select *
2 from Persona P
3 where P.Nome = P1.Nome and
4       P.Cognome = P1.Cognome and
5       P.CodFiscale <> P1.CodFiscale

```

- Si vogliono estrarre le persone che non hanno degli omonimi:

– Utilizzando una *query* nidificata:

```

1 select *
2 from Persona P
3 where not exists (select *
4                     from Persona P1
5                     where P1.Nome = P.Nome and
6                           P1.Cognome = P.Cognome and
7                           P1.CodFiscale <> P.CodFiscale)

```

– Utilizzando il costruttore di tupla:

```

1 select *
2 from Persona P
3 where (Nome ,Cognome) not in
4       (select Nome, Cognome
5            from Persona Q
6            where Q.CodFiscale <> P.CodFiscale)

```

Un altro **esempio**, si consideri il seguente schema:

Cantante (Nome, Canzone) Autore (Nome, Canzone)

- Si vogliono estrarre i cantautori puri, ovvero i cantanti che hanno eseguito solo canzoni di cui erano anche autori:

```
1 select Nome
2 from Cantante
3 where Nome not in
4     (select Nome
5      from Cantante C
6      where Nome not in
7          (select Nome
8             from Autore
9             where Autore.Canzone = C.Canzone))
```

La prima interrogazione nidificata (quella con `from Cantante C`) non ha alcun legame con l'interrogazione esterna, e può quindi essere eseguita in modo del tutto indipendente. Invece, l'interrogazione al livello successivo presenta un legame (`Autore.Canzone = C.Canzone`).

L'**esecuzione** dell'interrogazione avviene seguendo queste **fasi**:

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

13.22.4 Interrogazioni nidificate nelle clausole select e from

La sintassi SQL consente l'uso di interrogazioni nidificate anche come elemento della clausola **select** e nella clausola **from**.

Nella **clausola select** è possibile utilizzare interrogazioni nidificate per rappresentare interrogazioni che altrimenti sarebbero rappresentate mediante l'uso di **join**.

Per **esempio**:

- Si vuole estrarre per ogni cantante il numero di canzoni di cui è autore:

```
1 select distinct Nome, NumCanzoni = (select count(*)
2                               from Autore A
3                               where A.Nome = C.Nome)
4 from Cantante C
```

L'interrogazione potrebbe essere formulata utilizzando una struttura di **join** e il costrutto di raggruppamento:

```
1 select distinct C.Nome, count(*) as NumCanzoni
2 from Cantante C join Autore A on C.Nome = A.Nome
3 group by C.Nome
```

In generale è necessaria **grande cautela** nell'uso di questo costrutto, in quanto è indispensabile che l'interrogazione nidificata restituisca un'unica tupla come risultato. Solitamente l'interrogazione nidificata presenta riferimenti nella sua clausola **where** a variabili dell'interrogazione stessa.

Nella **clausola from** è possibile **comporre catene di interrogazioni**, in cui ciascuna interrogazione utilizza il risultato della precedente come se fosse una tabella base.

Per **esempio**:

- Si vuole estrarre per ogni cantante che è anche autore di canzoni, il numero di canzoni di cui è rispettivamente interprete e autore:

```
1 select distinct C.Nome, NumCantante, NumScritte
2 from (select Nome, count(*) as NumCantante
3        from Cantante) C join
4      (select Nome, count(*) as NumScritte
5        from Autore) A
6      on C.Nome = A.Nome
```

Questa interrogazione assume semplicemente che vengano dapprima eseguita le interrogazioni nella clausola **from** e che i risultati di queste interrogazioni vengano utilizzate come normali tabelle.

13.23 Interrogazioni di tipo insiemistico (`union`, `intersect`, `except`)

In SQL esistono degli **operatori insiemistici** simili a quelli dell'algebra relazionale: unione (`union`), intersezione (`intersect`) e differenza (`except` o `minus`). Si noti che gli operatori `intersect` ed `except` possono essere espressi utilizzando altri costrutti. La loro sintassi è la seguente:

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

Gli operatori insiemistici **eseguono di default l'eliminazione dei duplicati**. Questo accade per due ragioni specifiche:

- L'eliminazione dei duplicati rispetta il tipico significato di questi operatori;
- L'esecuzione degli operatori insiemistici richiede un'analisi delle righe e l'aggiunta dell'eliminazione dei duplicati rende il costo addizionale molto limitato.

Nel caso in cui si vogliano **preservare tutti i duplicati**, sarà sufficiente utilizzare l'operatore insiemistico con la **parola chiave all**.

A differenza dell'algebra relazionale, gli operatori insiemistici non richiedono che gli schemi su cui vengono effettuate le operazioni siano identici. Tuttavia, **gli attributi devono essere di pari numero e con domini compatibili**. In SQL, con gli operatori insiemistici, la **corrispondenza tra gli attributi si basa sulla posizione** di essi.

Per **esempio**, si considerino le seguenti *query* che si riferiscono alla base dati a pagina 159:

- Si vogliono estrarre i nomi e i cognomi degli impiegati:

```
1 select Nome
2 from Impiegato
3      union
4 select Cognome
5 from Impiegato
```

I primi valori ottenuti sono quelli del **Nome** e successivamente quelli del **Cognome**. La tabella risultato è ottenuta unendo i due risultati. Come si vede dal seguente risultato, non vi sono duplicati grazie all'operatore insiemistico: *colonna Nome; valori Mario, Carlo, Giovanni, Franco, Lorenzo, Paola, Marco, Rossi, Bianchi, Verdi, Neri, Gialli, Rosati*.

- Si vogliono estrarre i nomi e i cognomi di tutti gli impiegati, eccetto quelli appartenenti al dipartimento Amministrazione, mantenendo i duplicati:

```

1 select Nome
2 from Impiegato
3 where Dipart <> 'Amministrazione'
4      union all
5 select Cognome
6 from Impiegato
7 where Dipart <> 'Amministrazione'
```

In questo caso tutti i duplicati vengono mantenuti e il risultato è: *colonna Nome; valori* Carlo, Franco, Carlo, Lorenzo, Marco, Bianchi, Neri, Rossi, Gialli, Franco.

- Si vogliono estrarre i cognomi di impiegati che sono anche nomi:

```

1 select Nome
2 from Impiegato
3      intersect
4 select Cognome
5 from Impiegato
```

Il risultato è un solo valore nella tabella Nome: Franco.

- Si vogliono estrarre i nomi degli impiegati che non sono cognomi di qualche impiegato:

```

1 select Nome
2 from Impiegato
3      except
4 select Cognome
5 from Impiegato
```

13.24 Viste (create view)

Le **viste** sono **tabelle “virtuali” il cui contenuto dipende dal contenuto delle altre tabelle in una base di dati**. In SQL vengono definite associando un nome e una lista di attributi al risultato dell'esecuzione di un'interrogazione. Nell'interrogazione che definisce la vista possono comparire anche altre viste:

```
1 create view NomeVista [ ( ListaAttributi ) ] as SelectSQL
```

L'interrogazione SQL deve restituire un insieme di attributi compatibile con gli attributi nello schema della vista; l'ordine nella clausola `select` deve corrispondere all'ordine degli attributi nello schema.

SQL **non consente** alcune operazioni con le viste:

- Dipendenze **immediate**, ovvero definire una vista in termini di sé stessa;
- Dipendenze **ricorsive**, ovvero definire un'interrogazione di base e un'interrogazione ricorsiva;
- Dipendenze **transitive circolari**, ovvero una vista che invoca la seconda vista, la seconda che invoca la terza, finché l'ultima non invoca di nuovo la prima (V_1 invoca V_2 , V_2 invoca V_3 , ..., V_n invoca V_1).

Lo **scopo** delle viste è quello di aumentare il potere espressivo del linguaggio. Per esempio, nella clausola `from` sono ammesse interrogazioni nidificate solo grazie alle viste.

Per **esempio**, si definiscono le seguenti *query* considerando la base dati a pagina 159:

- Si definisce una vista `ImpiegatiAmmin` che contiene tutti gli impiegati del dipartimento Amministrazione con uno stipendio superiore a 10:

```
1 create view ImpiegatiAmmin(Matricola, Nome,
                                Cognome, Stipendio) as
2 select Matricola, Nome, Cognome, Stipendio
3 from Impiegato
4 where Dipart = 'Amministrazione' and
5       Stipendio > 10
```

E si costruisce una relativa vista `ImpiegatiAmminPoveri` definita a partire dalla visita `ImpiegatiAmmin`, la quale conterrà gli impiegati amministrativi con uno stipendio compreso tra 10 e 50:

```
1 create view ImpiegatiAmminPoveri as
2 select *
3 from ImpiegatiAmmin
4 where Stipendio < 50
```

- Si vuole determinare qual è il dipartimento che spede il massimo in stipendi. A questo scopo, si definisce una visita:

```

1 create view BudgetStipendi(Dip, TotaleStipendi) as
2 select Dipart, sum(Stipendio)
3 from Impiegato
4 group by Dipart

```

E quindi si prosegue cercando di estrarre il dipartimento avente il valore massimo della somma degli stipendi:

```

1 select Dip
2 from BudgetStipendi
3 where TotaleStipendi = (select max(TotaleStipendi)
4                           from BudgetStipendi)

```

- Si crea la seguente vista:

```

1 create view DipartUffici(NomeDip, NroUffici) as
2 select Dipart, count(distinct Ufficio)
3 from Impiegato
4 group by Dipart

```

E si vuole estrarre il numero medio di uffici per ogni dipartimento:

```

1 select avg(NroUffici)
2 from DipartUffici

```

13.24.1 Esempio ampio con le viste

Il seguente schema relazionale descrive il calendario di una manifestazione sportiva a squadre nazionali:

Stadio (Nome, Citta, Capienza)

Incontro (NomeStadio, Data, Ora, Squadra1, Squadra2)

Nazionale (Paese, Continente, Categoria)

Si esegue la seguente *query*: si trova lo stadio in cui la squadra italiana gioca più partite. Ci sono due alternative:

1. L'opzione con la vista apposita:

```
1 create view StadiItalia(NomeStadio, NroPart) as
2 select NomeStadio, count(*) as NroPart
3 from Incontro
4 where Squadra1 = 'Italia' or
5       Squadra2 = 'Italia'
6 group by NomeStadio;
7
8 select Citta
9 from Stadio
10 where NomeStadio in
11      (select NomeStadio
12       from StadiItalia
13       where NroPart =
14           (select max(NroPart)
15            from StadiItalia))
```

2. L'opzione con la vista più generale:

```
1 create view Stadi(NomeStadio, Squadra, NroPart) as
2 select NomeStadio, Paese,
3        count(distinct Data, Ora) as NroPart
4 from Incontro, Nazionale
5 where (Squadra1 = Paese or Squadra2 = Paese)
6 group by NomeStadio, Paese;
7
8 select Citta
9 from Stadio
10 where NomeStadio in
11      (select NomeStadio
12       from Stadi
13       where Squadra = 'Italia' and
14           NroPart =
15           (select max(NroPart)
16            from Stadi
17            where Squadra = 'Italia'))
```

Il seguente schema relazionale descrive le moto e il suo relativo proprietario:

Moto (Targa, Cilindrata, Marca, Nazione, Tasse)

Proprietario (Nome, Targa)

Si esegue la seguente *query*: si vuole estrarre per ogni cliente le tasse che devono essere pagate per tutte le moto possedute, ipotizzando che se vi sono più proprietari per una moto, l'ammontare delle tasse viene equamente diviso tra i proprietari.

```
1 create view TasseInd(Targa, Tassa) as
2 select Targa, Tasse/count(*)
3 from Moto join Proprietario
4     on Moto.Targa = Proprietario.Targa
5 group by Targa, Tasse;
6
7 select Nome, sum(Tassa)
8 from Proprietario join TasseInd
9     on Proprietario.Targa = TasseInd.Targa
10 group by Nome
```