

BASI DI DATI



Indice

Introduzione (Capitolo 1)

1.1 Sistemi informativi, informazioni e dati

1.2 Basi di dati e DBMS

1.3 Modelli dei dati

 1.3.1 Schemi e Istanze

 1.3.2 Livelli di astrazione nei DBMS

1.4 Linguaggi e utenti delle basi di dati

 1.4.1 Utenti e progettisti

1.5 Vantaggi e svantaggi dei DBMS

Progettazione concettuale di DB (Capitolo 2)

2.1 Introduzione alla progettazione

2.2 La raccolta e l'analisi dei requisiti

2.3 Metodologie di progettazione

2.4 Progettazione concettuale

 2.4.1 Astrazione nella progettazione concettuale dei Database

 2.4.2 Criteri generali di rappresentazione

2.5 Modello Entità-Relazione

 2.5.1 Entità

 2.5.2 Relazione

 2.5.3 Attributi

 2.5.4 Generalizzazione e Specializzazione (modello EER)

2.6 Cardinalità delle relazioni

 2.6.1 Cardinalità degli attributi

2.7 Identificatori (interni) delle entità

 2.7.1 Identificatori (esterni) delle entità

2.8 Reificazione

2.9 Documentazione di schemi E-R

 2.9.1 Business rules

 2.9.2 Tecniche di documentazione

2.10 Qualità dei modelli concettuali

2.11 Strategie di progetto

Progettazione Logica di DB (Capitolo 3)

- 3.1 Introduzione
- 3.2 Fasi della progettazione logica
- 3.3 Analisi delle prestazioni sullo schema E-R
- 3.4 Ristrutturazione di schemi E-R
- 3.5 Traduzione verso il modello relazionale
 - Modello uno a molti
 - Modello uno a uno
 - Modello molti a molti

Modello relazionale (Capitolo 4)

- 4.1 Struttura modello relazionale
 - 4.1.1 Relazioni e tabelle
- 4.2 Relazioni e basi di dati
- 4.3 Informazione incompleta e valori nulli
- 4.4 Vincoli di integrità
 - 4.4.1 Chiavi
 - 4.4.2 Chiavi e valori nulli
 - 4.4.3 Vincoli di integrità referenziali

Algebra e calcolo relazionale (Capitolo 5)

- 5.1 Linguaggi per database
- 5.2 Algebra relazionale
 - 5.2.1 Operatori insiemistici
 - 5.2.2 Ridenominazione
 - 5.2.3 Selezione
 - 5.2.4 Proiezione
 - 5.2.5 Prodotto cartesiano
 - 5.2.6 Join
- 5.3 Equivalenza di espressioni
- 5.4 Algebra con valori nulli

SQL (Capitolo 6)

6.1 Introduzione

6.2 I domini elementari

 6.2.1 Definizione delle tabelle (create table)

 6.2.2 Definizione dei domini (create domain)

 6.2.3 Specifica di valori di default

 6.2.4 Vincoli intrarelazionali

 6.2.5 Vincoli interrelazionali

 6.2.6 Modifica degli schemi

 6.2.7 Cataloghi relazionali

6.3 Query in SQL

 6.3.1 Query semplici

 6.3.2 Gestione dei valori nulli

 6.3.3 Duplicati

 6.3.4 Join interni ed esterni

 6.3.5 Uso di variabili

 6.3.6 Ordinamento (order by)

 6.3.7 Operatori aggregati

 6.3.8 Interrogazioni con raggruppamento (group by)

 6.3.9 Clausola having

 6.3.10 Query di tipo insiemistico

6.4 Query nidificate

 6.4.1 (not) exists

6.5 Modifica dei dati in SQL

6.6 Operatore between

SQL avanzato (Capitolo 7)

7.1 Vincoli di integrità generici

 7.1.1 Asserzioni

 7.1.2 Viste

 7.1.3 Viste ricorsive

7.2 Controllo dell'accesso

 7.2.1 Risorse e privilegi

 7.2.2 Comandi per concedere e revocare i privilegi

Normalizzazione (Capitolo 8)

8.1 Introduzione

8.2 Dipendenza funzionale

8.3 Forma normale Boyce e Codd

 8.3.1 Decomposizione senza perdita

 8.3.2 Conservazione delle dipendenze

 8.3.3 Qualità delle decomposizioni

8.4 Terza forma normale

8.5 Prima forma normale

8.6 Seconda forma normale



Database – Basi di dati

Introduzione

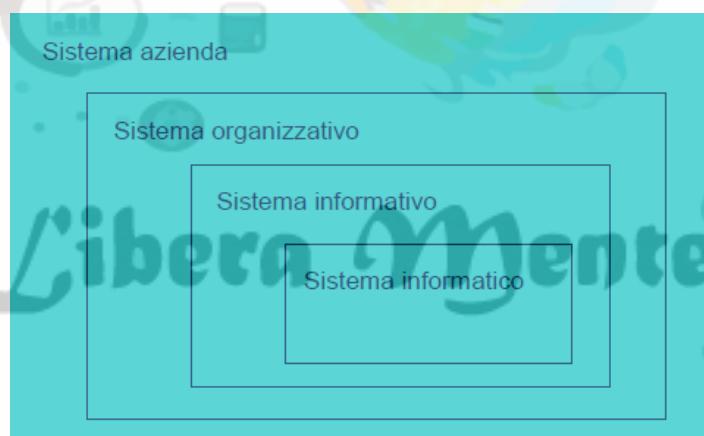
1.1 Sistemi informativi, informazioni e dati

Nello svolgimento di ogni attività è essenziale la disponibilità di informazioni e la capacità di gestirle in modo efficace: ogni organizzazione è dotata di un **sistema informativo**, una componente (sottosistema) che organizza e gestisce le informazioni necessarie per seguire gli scopi dell'organizzazione stessa. Ogni organizzazione ha un sistema informativo, eventualmente non esplicitato nella struttura. Quasi sempre, il sistema informativo è di supporto ad altri sottosistemi e di solito è suddiviso in sottosistemi (in modo gerarchico o decentrato), più o meno fortemente integrati.

Il **sistema organizzativo** è l'insieme di risorse (persone denaro, materiali, informazioni) e regole per lo svolgimento delle attività al fine di raggiungere degli scopi. Il sistema informativo è parte del sistema organizzativo, infatti, esegue/gestisce processi che coinvolgono informazioni.

I sistemi informativi esistono da molto prima dell'invenzione dei calcolatori elettronici, per indicare la porzione automatizzata del sistema informativo viene utilizzato il termine di **sistema informatico**.

Nei sistemi informatici le **informazioni** vengono rappresentate per mezzo di **dati**, che hanno bisogno di essere interpretati per fornire informazioni.



1.2 Basi di dati e DBMS

I sistemi software dedicati alla gestione dei dati sono stati realizzati solo a partire dalla fine degli anni 60, in assenza di un software specifico, la gestione dei dati è affidata ai linguaggi di programmazione come il *C* e il *Fortran*, oppure ad alcuni linguaggi ad oggetti quali *C++* e *Java*. I dati però sono replicati tante volte quanti sono i programmi che li utilizzano, con diverse ridondanze.

Le **basi di dati** sono state progettate per superare questo tipo di inconvenienti, gestendo in modo più semplificato le informazioni di interesse per diversi soggetti, riducendo le ridondanze. Una base di dati è una collezione di dati per una o più applicazioni.

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

- *Grandi*: hanno dimensioni maggiori della memoria centrale dei sistemi di calcolo utilizzati e il limite deve essere solo quello fisico dei dispositivi.

- *Condivise*: applicazioni utenti diversi devono poter accedere a dati comuni. Per garantire l'accesso condiviso, il dispone di un meccanismo chiamato *controllo di concorrenza*.
- *Persistenti*: hanno un tempo di vita indipendente, infatti, non dipendono dalle esecuzioni dei programmi che le utilizzano. Per questo i dati gestiti da un programma in memoria centrale hanno una vita che inizia e termina con l'esecuzione del programma e tali dati non sono persistenti.
- *Affidabili*: cioè la capacità di conservare il contenuto delle basi di dati in caso di malfunzionamento hardware e software. A questo scopo i DBMS forniscono funzionalità di salvataggio e ripristino (*backup e recovery*).

Un DBMS deve esse, inoltre, *efficiente* ed *efficace*. Quindi una base di dati è una collezione di dati gestita da un DBMS.

La gestione di collezioni di dati grandi è possibile anche per mezzo di file. Questi sono stati introdotti per gestire insiemi di dati “localmente” a una procedura o applicazione. I DBMS sono stati concepiti e realizzati per estendere le funzioni dei ***file system***, fornendo la possibilità di accesso condiviso ai dati da parte di più utenti applicazioni.



1.3 Modelli dei dati

Un ***modello dei dati*** è un insieme di concetti utilizzati per organizzare i dati di interesse e descriverne la struttura in modo che esso risulti comprensibile ad un elaboratore. Per esempio, il C permette di costruire tipi per mezzo di costruttori *struct, union, enum, * (pointer)*.

Il ***modello relazionale*** dei dati, attualmente il più diffuso, permette di definire tipi per mezzo del *costruttore relazionale*, che consente di organizzare i dati in insiemi di record a struttura fissa. La relazione viene rappresentata per mezzo di una tabella, le cui righe rappresentano i record e le colonne corrispondono ai campi dei record; l'ordine delle righe delle colonne è irrilevante.

Oltre al modello relazionale sono stati definiti altri 4 tipi di modelli:

- 1) Il *modello gerarchico*, basato sull'uso di struttura ad albero definito durante la prima fase di sviluppo dei DBMS.
- 2) il *modello reticolare* (detto anche *CODASYL*) basato sull'uso di grafi.
- 3) il *modello oggetti* che stende alle basi di dati il paradigma di programmazione oggetti.
- 4) il *modello XML* dove i dati vengono presentati insieme alla loro descrizione e non devono sottostare ad un'unica struttura logica.
- 5) *modelli semistrutturati e flessibili* sviluppati i sistemi *NoSQL*, cercano di superare alcune limitazioni dei sistemi relazionali.

I **modelli** dei dati appena elencati vengono detti *logici*, per sottolineare il fatto che riflettono una particolare organizzazione (ad alberi, a grafi, a tabelle o a oggetti). Più recentemente sono stati introdotti altri modelli dei dati detti **concettuali**, utilizzati per descrivere i dati in maniera indipendente dal modello logico. Il loro nome deriva dal fatto che tendono a descrivere i *concetti* del mondo reale. Essi vengono utilizzati nella fase preliminare di progettazione di basi di dati.

1.3.1 Schemi e Istanze

Nelle basi di dati esiste una parte invariante nel tempo, detta **schema** della base di dati, costituita dalle caratteristiche dei dati quindi ne descrive la struttura, e una parte variabile nel tempo, detta **istanza** o stato della base di dati, costituita dai valori effettivi.

Lo schema è la parte **intensionale**, mentre l'istanza è la parte **estensionale**.

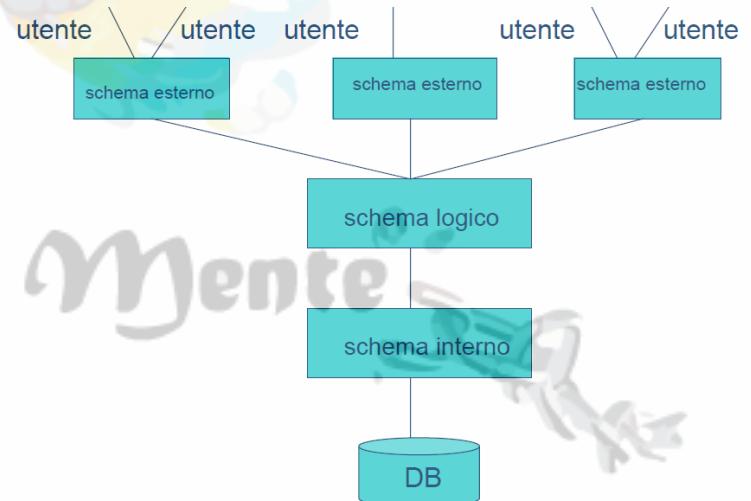
In fine diciamo che abbiamo una **relazione tra modelli, schemi e istanze** perché

- il *modello* fornisce le regole per strutturare i dati;
- lo *schema* verifica che una istanza è valida;
- *l'istanza* descrive la realtà in un dato istante di tempo.

1.3.2 Livelli di astrazione nei DBMS

L'architettura di una base di dati è organizzata su 3 livelli, detti *esterno*, *logico* e *interno*:

- Il livello **logico** (parte estensionale) costituisce una descrizione dell'intera base di dati per mezzo del modello logico adottato nei DBMS.
- Il livello **interno** costituisce la rappresentazione dello schema logico per mezzo di strutture fisiche di memorizzazione.
- Il livello **esterno** costituisce la descrizione di una porzione della base di dati per mezzo del modello logico.



Mediante questa definizione livelli è possibile ottenere ***l'indipendenza dei dati***, ovvero le applicazioni sono indipendenti dal modo in cui i dati sono organizzati. L'indipendenza dei dati può essere caratterizzata come indipendenza *fisica* e *logica*:

- ***l'indipendenza fisica*** consente di interagire con il DBMS in modo indipendente dalla struttura fisica dei dati
- ***l'indipendenza logica*** consente di interagire con il livello esterno della base di dati in modo indipendente dal livello logico. È possibile modificare il livello logico mantenendo inalterate le strutture esterne di interesse per l'utente.

1.4 Linguaggi e utenti delle basi di dati

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

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

- *linguaggi di definizione dei dati* o **Data Definition Language (DDL)**, utilizzati per definire gli schemi logici, esterni e fisici.
- *linguaggi di manipolazione dei dati* o **Data Manipulation Language (DML)** utilizzati per l'inserimento, la cancellazione e la modifica dei dati.

1.4.1 Utenti e progettisti

Varie categorie di persone possono interagire con una base di dati o con un DBMS.

- **L'amministratore della base di dati** (*Database Administrator – DA*) è la persona responsabile del controllo della progettazione e amministrazione della base di dati
- I **progettisti e programmati** di applicazioni implementano programmi che accedono alla base di dati.
- Gli **utenti** utilizzano la base di dati per le proprie attività. Essi sono divisi in due categorie:
 1. **utenti finali** (o *terminalisti*), utilizzano programmi che realizzano attività standard predefinite.
 2. **utenti casuali** in grado di utilizzare i DML per l'accesso alla base di dati.

1.5 Vantaggi e svantaggi dei DBMS

Elenchiamo i seguenti **vantaggi**:

- La condivisione permette la riduzione di ridondanze.
- L'indipendenza dei dati favorisce lo sviluppo e la manutenzione delle applicazioni.
- Controllo centralizzato dei dati.
- Disponibilità di servizi integrati.
- Dati come risorsa comune.

Gli **svantaggi**, invece, sono:

- Costo dei prodotti e della transizione elevato.

Progettazione Concettuale di DB

2.1 Introduzione alla progettazione

Progettare una base di dati significa definirne la struttura, le caratteristiche e il contenuto. La progettazione di una base di dati costituisce solo una delle diverse fasi di sviluppo di un sistema informativo. Il **ciclo di vita** di un sistema informativo comprende:

- **Studio di fattibilità.** Serve a definire i costi delle varie alternative e a stabilire le proprietà delle varie componenti del sistema.
- **Raccolta e analisi dei requisiti.** E' fatta nella *progettazione concettuale* e consiste nello studio delle proprietà e delle funzionalità del sistema informativo. Questa fase richiede un'interazione con gli utenti e verranno, inoltre, stabiliti i requisiti software e hardware del sistema informativo. Le attività principali svolte in questa fase sono la costruzione di un **glossario dei termini**, l'eliminazione delle ambiguità (sinonimi) e il raggruppamento dei requisiti "omogenei".
- **Progettazione.** Si divide in *progettazione dei dati* e *progettazione delle applicazioni*. Nella prima si individua la struttura e l'organizzazione che i dati dovranno avere, nell'altra si definiscono le caratteristiche dei programmi applicativi.
- **Implementazione.** Viene costruito e popolata la base di dati e viene prodotto il codice dei programmi.
- **Validazione e collaudo.** Serve a verificare il corretto funzionamento e la qualità del sistema informativo.
- **Funzionamento.** In questa fase il sistema informativo diventa operativo e se non si verificano malfunzionamenti questa attività richiede solo operazioni di manutenzione.

Inoltre, oltre a queste fasi ce ne potrebbe essere un'altra quella di *prototipizzazione*, che consiste nella realizzazione di una versione semplificata del sistema informativo.



2.2 La raccolta e l'analisi dei requisiti

Per **raccolta dei requisiti** si intende l'individuazione dei problemi che l'applicazione da realizzare deve risolvere e le caratteristiche che deve assumere. Per **caratteristiche del sistema** si intendono sia gli **aspetti statici** (*i dati*) sia gli **aspetti dinamici** (*le operazioni sui dati*). I requisiti vengono espressi in linguaggio naturale e per questo motivo possono essere ambigui e disorganizzati.

L'**analisi dei requisiti** consiste nel chiarimento e nell'organizzazione delle *specifiche dei requisiti*. I requisiti di un'applicazione provengono da fonti diverse che possono essere:

- **Gli utenti dell'applicazione:** in questo caso le informazioni si acquisiscono attraverso alcune interviste.
- **La documentazione esistente;**
- **Eventuali realizzazioni preesistenti:** ovvero applicazioni che devono interagire con il sistema di realizzare.

Nella **fase di acquisizione delle specifiche** è importante l'interazione con gli utenti del sistema informativo. Durante questa fase può venire che gli utenti diversi forniscono informazioni diverse, è opportuno quindi effettuare con l'utente diverse verifiche di comprensione fatte attraverso l'utilizzo di esempi. Le regole generali, per ottenere una *specifica dei requisiti* più precisa e senza ambiguità, sono:

- **Scegliere il corretto livello di astrazione:** bisogna evitare di utilizzare termini troppo generici o troppo specifici che rendono poco chiaro un concetto.
- **Standardizzare la struttura delle frasi:** è preferibile utilizzare sempre lo stesso stile sintetico, per esempio “*per <dato> rappresentiamo <insieme di proprietà>*”.
- **Evitare le frasi contorte.**
- **Individuare sinonimi e unificare i termini.**
- **Rendere esplicito il riferimento tra termini.**
- **Costruire un glossario dei termini** che contenga: una breve descrizione, possibili sinonimi e altri termini contenuti nel glossario con il quale esiste un legame logico.

2.3 Metodologie di progettazione

Per progettare una base di dati di buona qualità è opportuno seguire una **metodologia di progettazione** che consiste in:

- una **decomposizione** dell'intera attività in passi successivi indipendenti tra loro;
- una **serie di strategie** da seguire nei vari passi e alcuni criteri di scelta quando ci sono più alternative;
- alcuni **modelli di riferimento** per descrivere i dati I/O delle fasi.

Le proprietà che una metodologia deve garantire sono:

- la **generalità** rispetto alle applicazioni e ai sistemi;
- la **qualità** del prodotto in termini di *correttezza, completezza ed efficienza* rispetto alle risorse;
- la **facilità d'uso** delle strategie e dei modelli.

Negli anni sia è affermata una metodologia di progetto che ha soddisfatto pienamente le proprietà descritte. Questa metodologia è articolata in tre fasi: la prima fase indica “**cosa**” rappresentare in una base dati, la seconda e terza fase stabilisce “**come**” fare una base dati.

- 1) Progettazione concettuale.** Il prodotto di questa fase viene chiamato **schema concettuale**, cioè una descrizione formale delle esigenze aziendali, espressa in modo indipendente dal DBMS adottato.
Lo schema concettuale fa riferimento a un **modello concettuale**, un linguaggio ad alto livello usato per descrivere lo schema concettuale.
In questa fase il progettista deve cercare di rappresentare l'informazione della base dati, infatti, la fase di raccolta e analisi dei requisiti viene svolta insieme a quella di progettazione concettuale.
- 2) Progettazione logica.** Consiste nella traduzione dello schema concettuale definito nella fase precedente. Il prodotto di questa fase viene chiamato **schema logico**, espresso nel DDL del DBMS, e fa riferimento ad un **modello logico** (linguaggio usato per descrivere lo schema logico) dei dati.
In questa fase si considerano aspetti legati all'efficienza e ai vincoli.
- 3) Progettazione fisica.** In questa fase si operano scelte strettamente dipendenti dallo specifico DBMS utilizzato, inoltre, lo schema logico viene completato con la memorizzazione dei dati. Il prodotto di questa fase viene chiamato **schema fisico**, che descrive le strutture di memorizzazione e accesso ai dati e fa riferimento ad un **modello fisico** dei dati.

2.4 Progettazione concettuale

La **progettazione concettuale** di una DB consiste nella costruzione di uno schema **Entità Relazionale** in grado di descrivere al meglio le specifiche sui dati di un'applicazione.

Perché la progettazione concettuale?

1. Perché la qualità dello schema migliora;
2. Il progetto converge verso risultati attesi;
3. I costi di sviluppo diminuiscono;
4. La scelta del DBMS è posticipata;
5. Lo schema concettuale può essere usato come punto di partenza per un nuovo progetto;
6. Differenti database possono essere confrontati in una struttura omogenea.

2.4.1 Astrazione nella progettazione concettuale dei Database

L'**astrazione** è un procedimento mentale che permette di evidenziare le proprietà più importanti degli oggetti osservati, escludendone quelle non rilevanti. Nella progettazione concettuale ci sono 3 tecniche di astrazione:

- 1) **Astrazione per classificazione.** A partire dalle singole persone e considerando le proprietà che le accomunano, si giunge al concetto di *PERSONA*, cioè alla definizione di una classe, che rappresenta un insieme di oggetti (le singole persone).
- 2) **Astrazione per aggregazione.** Si può giungere alla definizione di una classe attraverso l'astrazione di un insieme di parti o proprietà. Ad esempio, definite le classi *NOME*, *COGNOME*, *ETÀ*, *SESSO*, *STIPENDIO*, una loro astrazione di aggregazione è la classe *PERSONA*.
- 3) **Astrazione per generalizzazione.** Si può giungere alla definizione di una classe come unione di un insieme di classi, ognuna delle quali è contenuta nella classe data. Ad esempio, definite le classi *UOMO* e *DONNA*, possiamo definire come loro generalizzazione la classe *PERSONA*.

2.4.2 Criteri generali di rappresentazione

Durante la progettazione concettuale è buona norma seguire alcune “regole concettuali” del modello E-R:

- Se un concetto ha proprietà importanti e/o descrive classi di oggetti autonome, è opportuno rappresentarlo con un’**entità**.
- Se un concetto ha una struttura semplice e non possiede proprietà importanti associate, è opportuno rappresentarlo come un **attributo** di un altro concetto a cui si riferisce.
- Se sono state individuate due o più entità e nei requisiti compare un concetto che li associa, questo concetto può essere rappresentato da una **relazione**.
- Se uno o più concetti sono dettagli di un concetto generale, è opportuno rappresentarli attraverso una **generalizzazione**.

2.5 Modello Entità-Relazione

Il modello **Entità-Relazione** è un **modello concettuale** di dati e fornisce una serie di strutture, dette **costrutti**, usate per descrivere la realtà di interesse in maniera facile da comprendere.

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

Per ogni costrutto, esiste una relativa rappresentazione grafica. Mediante tale modello è possibile costruire un grafo, ossia uno schema che costituisce una fotografia del fenomeno. Ciascuno schema è il risultato della composizione logica di 5 tipi di strutture di rappresentazione. Tali strutture sono: **l’entità**, **la relazione**, **l’attributo**, **i sottoinsiemi**, **la gerarchia di generalizzazione**.

2.5.1 Entità

Le **entità** rappresentano classi di oggetti che hanno proprietà comuni: *CITTA'*, *DIPARTIMENTO*, *IMPIEGATO*, *ACQUISTO* e *VENDITA* sono esempi di entità di un'applicazione aziendale.

Un'**occorrenza di entità** è un elemento della classe, è l’oggetto stesso (per esempio l’*IMPIEGATO* in carne e ossa). Un’occorrenza di entità ha un’esistenza e un’identità indipendentemente dalle proprietà che le vengono associate.

Ogni entità ha un nome che la identifica nello schema, inoltre, è conveniente utilizzare nomi espressivi e al singolare e usando sostantivi anziché verbi. Ogni entità viene rappresentata graficamente mediante un **rettangolo** con il nome dell’entità all’interno.

Quando riusciremo a trovare sempre un identificatore per un’entità, questa verrà chiamata **entità forte**. Nel momento in cui non riusciremo a trovare un identificatore o attributi per un’entità, questa viene chiamata **entità debole**. L’entità debole si appoggia ad un’altra entità.

2.5.2 Relazione

La **relazione** rappresenta un legame logico tra due o più entità. *FREQUENZA* è un esempio di relazione che si trova tra l’entità *STUDENTE* e *CORSO*.



In uno schema *E-R*, ogni relazione ha un nome che la identifica univocamente e viene rappresentata graficamente mediante un **rombo**.

Possono esistere relazioni diverse che coinvolgono la stessa entità. Nella scelta dei nomi di relazione è preferibile utilizzare i sostantivi invece che i verbi. Il **grado della relazione** indica il numero di entità che sono coinvolte.



È anche possibile avere relazioni ricorsive, ovvero una relazione che coinvolge due volte la stessa entità (**in alcuni casi è necessario specificare i “ruoli” delle entità**), oppure avere relazioni n-arie, c'è che coinvolgono più di due entità.

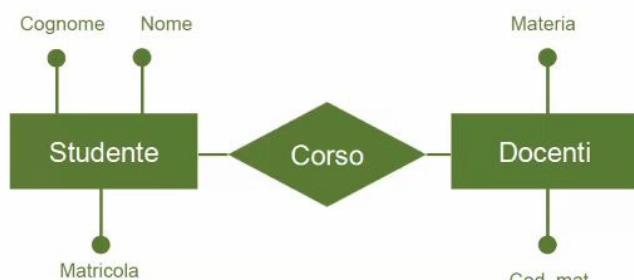


2.5.3 Attributi

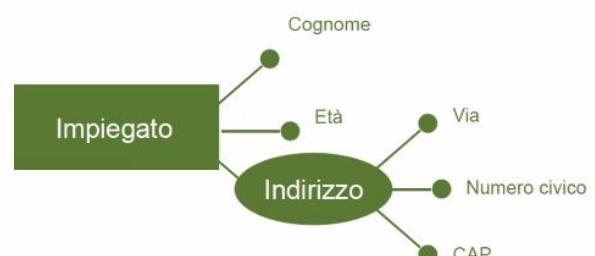
Sono rappresentati graficamente come una **riga** con un **pallino finale**. Gli **attributi** descrivono le proprietà elementari di entità o relazione. Ad ogni attributo è associato un insieme di valori, chiamato **dominio**. I domini non vengono riportati nello schema, ma vengono descritti nella documentazione associata.

Può risultare comodo raggruppare gli attributi in una stessa entità o relazione che presentano affinità nel loro significato o uso, si ottiene così un **attributo composto**, rappresentato in questo modo:

Per esempio, possiamo raggruppare gli attributi *VIA*, *Numero civico* e *CAP* dell'entità *PERSONA* per formare l'attributo *Indirizzo*.



Attributo atomico



Attributo composto

Infine, un **attributo multivaleore** può avere, per ogni istanza dell'entità associata, più valori. Ad esempio, un *attributo multivaleore* potrebbe essere *TELEFONO* per il quale possono essere noti più numeri; quindi un **attributo è multivaleore** se la sua cardinalità massima è pari a N. Un attributo viene rappresentato graficamente con un pallino tratteggiato, è detto **attributo derivabile**, ovvero un attributo che lo si può derivare sulla base di dati già esistenti: per esempio il *COD. FISCALE*, l'*INDIRIZZO* e la *DATA di NASCITA*.

2.5.4 Generalizzazione e Specializzazione (modello EER)

I diagrammi EER (*Enhanced Entity - Relationship*) sono una versione avanzata dei diagrammi ER. I modelli EER sono strumenti utili per la progettazione di database con modelli di alto livello. Con le loro funzionalità avanzate, è possibile pianificare i database in modo più approfondito andando, quindi, a trattare i particolari del DB.

Le **generalizzazioni** rappresentano legami logici tra un'entità E , detta **entità genitore**, e una o più entità $E_1 \dots E_n$, dette **entità figlie**. In questo caso 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*. Le generalizzazioni hanno le seguenti proprietà:

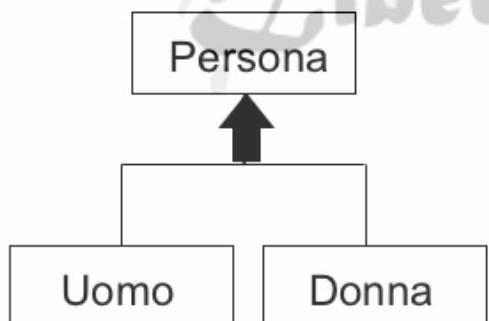
- Ogni occorrenza di un'entità figlia è anche una occorrenza dell'entità genitore;
- Ogni proprietà dell'entità genitore (*attributi, identificatori, relazioni e altre generalizzazioni*) è anche una proprietà delle entità figlie. Questa proprietà delle generalizzazioni si chiama **ereditarietà**.

Possono esserci generalizzazioni su più livelli: questo caso si chiama **gerarchia di generalizzazioni**.

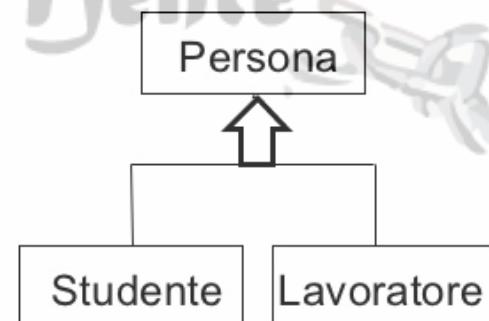
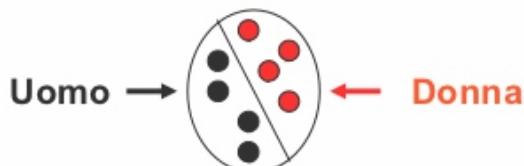
Due entità si trovano in una relazione di **sottoinsieme** se ogni istanza dell'*entità dipendente* (detta entità figlia) è anche istanza dell'*entità superiore* (detta entità padre), mentre non è vero il contrario.

Le generalizzazioni possono essere classificate in:

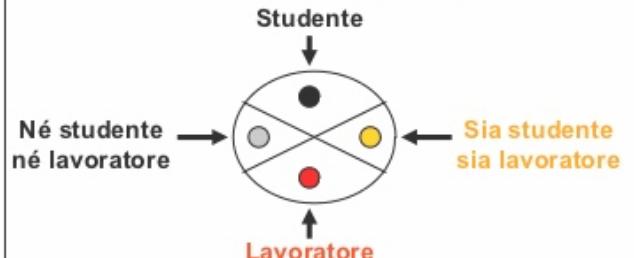
- **Totale** (*Si rappresenta con la freccia piena*), se ogni occorrenza dell'entità padre fa parte di una delle entità figlie, altrimenti è **parziale** (*Si rappresenta con la freccia vuota*).
- **Disgiunta (o esclusiva)**, se ogni occorrenza dell'entità padre è al più un'occorrenza di una delle entità figlie, altrimenti è **sovraposta**.



Totale ed esclusiva



Parziale e sovrapposta



2.6 Cardinalità delle relazioni

La **cardinalità delle relazioni** è una coppia di valori associati ad ogni entità che partecipa ad una *relationship*. La cardinalità specifica la partecipazione di un'entità a una relazione, il *primo numero* indica la **partecipazione**, il *secondo numero* indica la **cardinalità della relazione**.

È possibile assegnare qualunque intero non negativo a una cardinalità di relazione con l'unico vincolo che la cardinalità minima deve essere < o = della cardinalità massima. Per semplicità vengono utilizzati solamente tre simboli:

- **0 e 1 per la cardinalità minima:**

- 0, si dice che la *partecipazione* dell'entità relativa è *opzionale*;
- 1, si dice che la *partecipazione* è *obbligatoria*.

- **1 e N per la massima:**

- N non pone alcun limite.

Guardando le cardinalità massime coinvolte nella *relationship* è possibile stabilire il tipo di relazione (*uno-uno*, *uno-molti*, *molti-molti*).

Le relazioni aventi cardinalità massima pari a 1, per entrambe le entità coinvolte, definiscono una corrispondenza uno a uno e vengono quindi denominate **relazioni uno a uno**.

Le relazioni aventi un'entità con cardinalità massima pari a 1 e l'altra con cardinalità massima pari a N, sono denominate **relazioni uno a molti**.

Infine, le relazioni aventi cardinalità massima pari a N per entrambe le entità coinvolte, vengono denominate **relazioni molti a molti**.

2.6.1 Cardinalità degli attributi

La **cardinalità degli attributi** viene specificata per gli attributi di entità o relazioni e descrivono il numero minimo e massimo di valori dell'attributo associati a ogni occorrenza di entità o relazione. Nella maggior parte dei casi, la cardinalità di un attributo è pari a (1,1) e viene omessa.

Un **attributo** si dice **nullo** quando la *cardinalità minima* è pari a 0, oppure possono esistere diversi valori quindi avremo una *cardinalità massima* pari a molti (N) e avremo un **attributo multivale**.

- **cardinalità minima degli attributi può essere:**

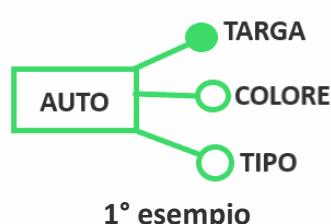
- 0, si dice che la *partecipazione* dell'entità relativa è *opzionale*;
- 1, si dice che la *partecipazione* è *obbligatoria*.

- **N per la cardinalità massima:**

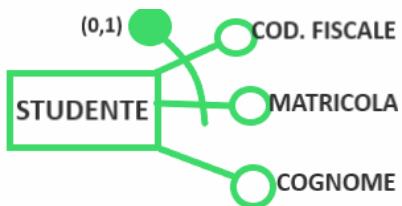
- N, *attributo multivale*.

2.7 Identificatori (interni) delle entità

Gli **identificatori delle entità** vengono utilizzati per ciascuna entità dello schema e descrivono gli attributi in modo da identificare univocamente le occorrenze delle entità. Sono detti **interni (o chiavi)** perché uno o più attributi sono sufficienti ad individuare un identificatore.



Nel **primo esempio**, la *TARGA* basta a identificare una ed una sola *AUTO*.

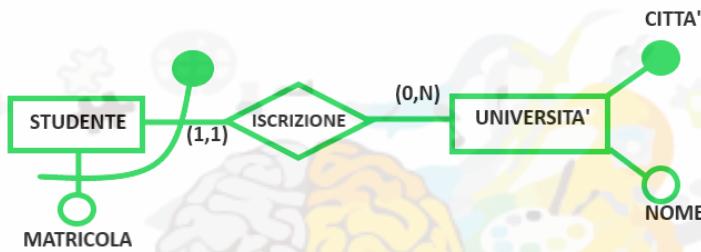


2° esempio

Nel **secondo esempio**, ci riferiamo a *STUDENTI* che possono avere la stessa *MATRICOLA*, occorre aggiungere come identificatore anche il *CODICE FISCALE* poiché in questo caso parliamo di Università diverse e potremmo trovare più matricole uguali.

2.7.1 Identificatori (esterni) delle entità

Nel caso in cui l'identificatore di un'entità è ottenuto utilizzando altre entità, allora si parla di **identificatore esterno**.



In questo esempio consideriamo *STUDENTI* provenienti da diversi atenei (le matricole possono essere uguali). Ogni studente può essere iscritto ad una sola *UNIVERSITA'*. La matricola quindi non identifica lo studente. L'attributo identificativo è in questo caso la matricola legata ad un'altra entità, *UNIVERSITA'*: quindi conosco lo studente se ne conosco la matricola e se so a quale università è iscritto.

REGOLE:

- 1) Un **identificatore** può coinvolgere uno o più attributi, ognuno dei quali deve avere cardinalità (1,1).
- 2) 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).
- 3) Un **identificazione esterna** può coinvolgere un'entità che a sua volta identificata esternamente, purché non vengano generati cicli di *identificazioni esterne*.
- 4) Ogni entità deve avere almeno un **identificare**, e ne può avere più di uno.
- 5) L'**identificatore** appartiene **SOLO** all'entità e **NON** alla relazione.

2.8 Reificazione

La **reificazione** è la trasformazione di un costrutto (*molti a molti*) in un'**entità debole**.

2.9 Documentazione di schemi E-R

Uno schema E-R non è quasi mai sufficiente, da solo, a rappresentare nel dettaglio tutti gli aspetti di un'applicazione. Risulta, infatti, indispensabile corredare ogni schema E-R con una **documentazione** di supporto, che possa servire a facilitare l'interpretazione dello schema stesso e a descrivere dei dati rappresentati che non possono essere espressi dai costrutti del modello.

2.9.1 Business rules

Uno degli strumenti più usati dagli analisti di sistemi informativi per la descrizione di proprietà di un'applicazione che non si riesce a rappresentare con modelli concettuali è quello delle **regole aziendali o business rules**.

Una regola aziendale può essere:

1. La **descrizione di un concetto**, ovvero la definizione precisa di un'entità, un attributo o di una relazione del modello E-R.
2. Un **vincolo d'integrità**, una proprietà che deve essere soddisfatta dalle istanze di una base di dati. Ogni vincolo può essere visto come un predicato che può assumere il valore vero o falso: l'istanza soddisfa il vincolo, se il predicato assume il valore vero, viceversa se assume valore falso.
3. Una **derivazione**, ovvero un concetto che può essere ottenuto attraverso un calcolo matematico o da altri concetti dello schema.

Per le regole del primo tipo è impossibile definire una sintassi precisa e si fa in genere ricorso a frasi in linguaggio naturale. Queste regole vengono rappresentate sotto forma di *glossari*, raggruppando le descrizioni in maniera opportuna (per esempio, per entità e per relazione).

Le regole che descrivono **vincoli di integrità** possono essere espresse sotto forma di **asserzioni atomiche**, ovvero affermazioni che devono essere sempre verificate. Per enunciare tali regole usiamo la forma:
 $< \text{concetto} > \text{ deve/non deve } < \text{espressione su concetti} >$

dove i concetti possono corrispondere o a concetti dello schema E-R, oppure a concetti derivabili da essi.

Le **derivazioni**, invece, possono essere espresse specificando le operazioni (aritmetiche o di altro genere) che permettono di ottenere il concetto derivato:

$< \text{concetto} > \text{ si ottiene } < \text{operazioni su concetti} >$

Quando lo schema concettuale viene tradotto in un database, le regole aziendali **non descrittive** (**vincoli e derivazioni**) vanno codificate. Per implementare le regole aziendali è possibile seguire diversi approcci:

- Utilizzare il linguaggio *SQL*;
- Usare *triggers* o *regole attive*;
- Utilizzare qualche linguaggio di programmazione.

2.9.2 Tecniche di documentazione

La **documentazione** dei vari concetti rappresentati in uno schema può essere prodotta facendo uso di un *dizionario dei dati*. Esso è composto da due tavole:

- la prima descrive l'entità dello schema con il nome, una definizione informale, elenco di tutti gli attributi e possibili identificatori.
- L'altra tabella descrive le relazioni con il nome, una loro descrizione informale, l'elenco degli attributi e l'elenco delle entità coinvolte insieme alla loro cardinalità di partecipazione.

L'uso del dizionario dei dati è importante nei casi in cui lo schema è complesso; risulta pesante specificare direttamente sullo schema tutti gli attributi di entità e relazioni. Si può far ricorso a una tabella, nella quale vengono elencate le varie regole, specificando la loro tipologia.

Entità	Descrizione	Attributi	Identificatore
Relazione	Descrizione	Entità coinvolte	Attributi

→ **Dizionario dei dati**

2.10 Qualità dei modelli concettuali

Nella costruzione di uno schema concettuale vanno garantite alcune proprietà generali:

Correttezza Uno schema concettuale è *corretto* quando utilizza i costrutti messi a disposizione dal modello concettuale di riferimento. Gli errori possono essere *sintattici* o *semantici*.

Completezza 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.

Leggibilità Uno schema concettuale è *leggibile* quando rappresenta i requisiti in maniera naturale e facilmente comprensibile. Per garantire questa proprietà è necessaria una scelta opportuna dei nomi da dare ai concetti.

Minimalità Uno schema 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*.

2.11 Strategie di progetto

Lo sviluppo di uno *schema concettuale* può essere considerato un processo di ingegnerizzazione e, com tale, risultano applicabili varie **strategie di progetto** utilizzate anche in altre discipline:

1. **Top-Down**: lo *schema concettuale* viene prodotto a partire da uno schema iniziale generale con pochi concetti molto astratti. Tramite una serie di raffinamenti successivi si arriva allo schema finale dettagliato. Tale tecnica è applicabile solo se si ha una visione completa della realtà. Nel passaggio da un livello di raffinamento a un altro, lo schema viene modificato facendo uso di alcune trasformazioni elementari chiamate **primitive di trasformazione top-down** e sono:
 - L'associazione degli attributi a un'entità o a una relazione;
 - La reificazione di un attributo o di un'entità;
 - La decomposizione di una relazione in due relazioni;
 - La generalizzazione di un'entità.
2. **Bottom-Up**: le specifiche iniziali sono suddivise in frammenti sempre più semplici fino a quando descrivono un frammento elementare della realtà. I vari schemi ottenuti vengono fusi fino ad ottenere lo schema concettuale finale che si ottiene attraverso le **primitive di trasformazione bottom-up** che sono:
 - L'introduzione di una nuova entità o relazione dall'analisi delle specifiche;
 - L'individuazione di una generalizzazione riferita ad un'entità;
 - L'aggregazione di alcuni attributi in un'entità o in una relazione;Questa strategia si adatta bene a situazioni in cui esiste un gruppo che si divide i problemi, ma la successiva integrazione presenta delle difficoltà.
3. **Inside-Out**: è un caso particolare del *bottom-up*, infatti, si individuano solo alcuni concetti importanti e da questi si procede verso quelli più lontani (si dice che si procede a “*macchia d'olio*”). Questa strategia non richiede integrazione.
4. **Mista**: cerca di combinare i vantaggi della strategia *top-down* con quelli del *bottom-up*, infatti possiamo suddividere un problema complesso in sotto problemi e procedere per raffinamenti.

Progettazione Logica di DB

3.1 Introduzione

L'obiettivo della **progettazione logica** è quello di costruire uno **schema logico** in grado di descrivere tutte le informazioni del progetto *E-R* (schema + documentazione) prodotto in fase di *progettazione concettuale*. Prima di passare allo *schema logico*, lo schema *E-R* va ristrutturato in modo da semplificare e ottimizzare il progetto. In sintesi:

- La **progettazione concettuale** ha lo scopo di rappresentare in modo accurato e naturale i dati d'interesse;
- La **progettazione logica** è la base per la realizzazione dell'applicazione e tiene conto delle prestazioni del progetto.

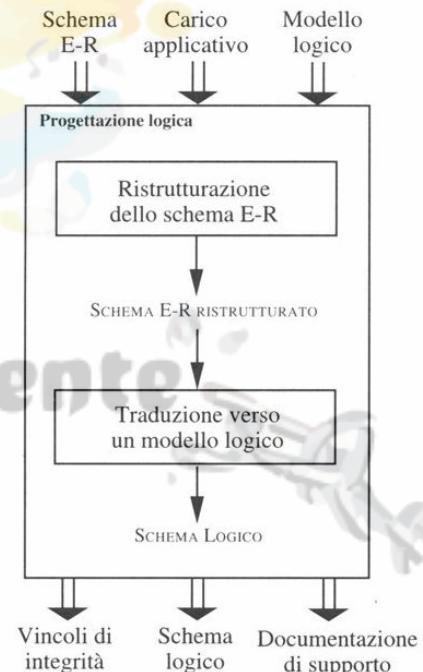
3.2 Fasi della progettazione logica

La **progettazione logica** è divisa in due fasi:

1. **Ristrutturazione dello schema E-R**: è una fase di ottimizzazione e semplificazione dello schema;
2. **Traduzione verso il modello logico**: include un'ulteriore ottimizzazione basandosi sempre sullo schema logico.

I dati in ingresso della **1°Fase** sono: lo *schema concettuale* prodotto nella fase precedente, e il **carico applicativo** che rappresenta il volume dei dati (# di occorrenze di entità e relazioni) e la loro dimensione, le opzioni e le loro caratteristiche.

Il risultato che si ottiene è uno schema *E-R* ristrutturato e insieme al modello logico formano i dati di ingresso della **2°Fase**. Il risultato è uno schema logico con la relativa documentazione.



3.3 Analisi delle prestazioni sullo schema E-R

Gli indici di prestazione di una base dati riguardano:

- ❖ **Costo di una operazione**: viene considerato il numero di occorrenze di entità e associazioni che vanno visitate per rispondere a un'operazione sulla base di dati.
- ❖ **Occupazione di memoria**: si valuta lo spazio di memoria (misurato in numero di byte) necessario per memorizzare i dati descritti dallo schema.

Per studiare questi parametri dobbiamo conoscere:

- **volume dei dati**, ovvero:
 - il numero di occorrenze di ogni entità associazione dello schema;
 - dimensioni di ciascun attributo (di entità o associazione).

- **caratteristiche delle operazioni:**

- tipo dell'operazione (*interattiva* o *batch*);
- frequenza (numero medio di esecuzione in un certo intervallo di tempo);
- dati coinvolti (entità e/o associazioni).

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

Nella **tavola dei volumi** vengono riportati tutti i **concetti** dello schema (entità e associazioni) con il **volumne** previsto. Se la cardinalità di un'entità è (1,1) allora la relazione assume lo stesso valore di tale entità.

Nella **tavola delle operazioni** viene riportato, per ogni **operazione**, la **frequenza** (il numero di volte che viene eseguita un'operazione) e un **tipo** che indica se l'operazione è:

- *interattiva* (*I*, c'è bisogno dei dati in input);
- oppure *batch* (*B*, non c'è la necessità dei dati in input, di solito l'operazione è batch quando troviamo la forma "Per ogni...").

Per ogni operazione, possiamo descrivere graficamente i dati coinvolti con uno **schema di operazione**, sul quale viene disegnato il "cammino logico" da percorrere nello schema E-R per accedere alle informazioni.

Infine, abbiamo la **tavola degli accessi**, che specifica se si tratta di entità o associazioni e se si fanno operazioni di **lettura** o **scrittura** e il **numero di accessi**. Le operazioni di scrittura sono più onerose di quelle in lettura e per convenzione una $1S = 2L$.

Tavola dei volumi

Concetto	Tipo	Volume
Sede	E	10
Composizione	E	80

Tavola delle operazioni

Operazione	Tipo	Frequenza
Op. 1	I	50 al giorno
Op. 2	I	100 al giorno

Tavola degli accessi

Concetto	Costrutto	Accessi	Tipo
Impiegato	Entità	1	L
Afferenza	Relazione	1	L

3.4 Ristrutturazione di schemi E-R

La fase di ristrutturazione si suddivide in diverse fasi:

1. Analisi delle ridondanze

In questa fase vengono analizzate tutte le **ridondanze** che corrispondono a un dato che viene derivato da altri dati, quindi si scelgono i dati da mantenere e quelli da eliminare.

Il vantaggio è una riduzione degli accessi necessari per calcolare il dato derivato; gli svantaggi sono una maggiore occupazione di memoria (spesso è un costo trascurabile) e la necessità di effettuare operazioni aggiuntive per tenere il dato aggiornato (un aggiornamento consiste sempre in una lettura e una scrittura, quindi 3 letture totali). La decisione di mantenere o eliminare una ridondanza va fatta sulla base delle operazioni che si vanno a fare sui dati ridondanti.

Si possono presentare varie forme di ridondanza:

- attributi derivabili da altri attributi della stessa entità;
- attributi derivabili dagli attributi di altre entità;
- attributi derivabile operazioni di conteggio di occorrenze

2. Eliminazione delle generalizzazioni

Dato che i sistemi tradizionali per la gestione delle basi dati non consente di rappresentare una generalizzazione, dobbiamo trasformare questo con costrutto in entità e associazioni. Abbiamo 3 alternative possibili:

1. **Accorpamento delle entità figlie nell'entità padre:** le entità E_1 ed E_2 vengono eliminate e le loro proprietà vengono aggiunte all'entità padre E_0 . Gli attributi delle entità figlie, A_{11} e A_{12} , possono assumere **valori nulli**, perché non importanti per l'entità padre E_0 . Questa soluzione può essere utile quando la specializzazione è **PARZIALE** e le entità figlie non hanno proprietà particolari. In questo caso, anche se abbiamo uno spreco di memoria per la presenza di *valori nulli*, avremo un numero minore di accessi.

Se la generalizzazione è totale la *cardinalità minima* sarà 1, se è parziale sarà 0.

Se la generalizzazione è sovrapposta la *cardinalità Massima* sarà N, se è esclusiva sarà 1.

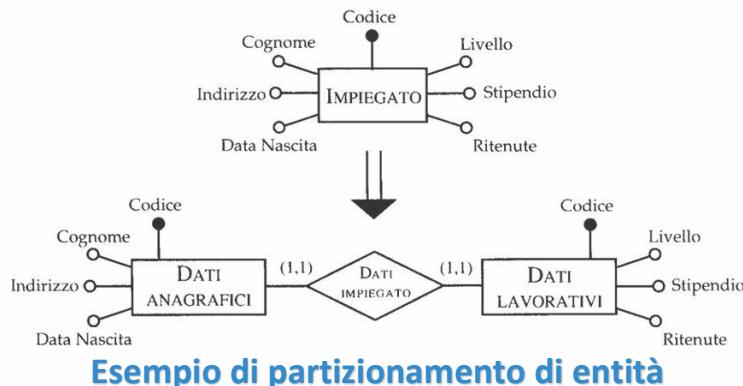
2. **Accorpamento del padre con le entità figlie:** l'entità genitore viene eliminata e, per la proprietà dell'ereditarietà, i suoi attributi, il suo identificatore e le relazioni a cui l'entità partecipava, vengono aggiunti alle entità figlie. È vantaggioso se le operazioni importanti fanno molta differenza tra le entità figlie. Questa soluzione è possibile solo se la generalizzazione è **TOTALE**. In questo caso le cardinalità non cambiano. In questo caso, avremo un risparmio della memoria perché gli attributi non assumono mai valori nulli e inoltre c'è una riduzione degli accessi dato che non si deve visitare E_0 .

3. **Sostituzione della generalizzazione con associazioni:** la generalizzazione si trasforma in due associazioni (1,1) che legano rispettivamente l'entità genitore con le entità figlie che diventano *entità deboli*. Non ci sono trasferimenti di attributi o associazioni; vanno aggiunti dei vincoli: ogni occorrenza di E_0 non può partecipare contemporaneamente alle due relazioni. Questa scelta è conveniente se la generalizzazione **NON** è **TOTALE** e se le operazioni fanno differenza tra entità padre e figlie. In questo caso, avremo un risparmio della memoria per l'assenza di valori nulli, ma avremo un incremento degli accessi. Le cardinalità cambiano, infatti avremo:
(1,1) per indicare l'esistenza di un'unica occorrenza;
(0,1) per indicare se si ha un'occorrenza di una o di un'altra entità.

3. Partizionamento/accorpamento di entità e associazioni

Le entità e le associazioni in uno schema E-R possono essere partizionati o accorpati per garantire una maggiore efficienza delle operazioni: gli accessi si riducono separando attributi di una stessa entità che vengono acceduti da operazioni diverse e raggruppando attributi di entità diverse che vengono accedute dalle stesse operazioni.

- **Partizionamento verticale:** si suddivide un'entità operando sui suoi attributi.
- **Partizionamento orizzontale:** la suddivisione avviene sulle occorrenze dell'entità. Per ogni operazione del genere si aggiunge una generalizzazione. I partizionamenti orizzontali hanno un problema, ovvero duplicano tutte le associazioni a cui l'entità originaria partecipa.



Esempio di partizionamento di entità

● Eliminazione Attributi Composti

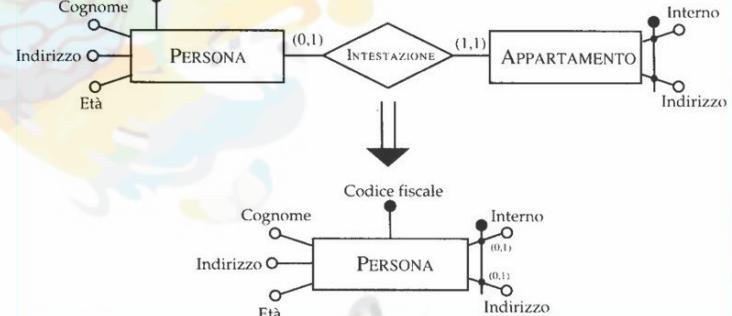
Si eliminano perché nello schema relazionale non esistono attributi composti

● Eliminazione Attributi Multivalore

Questa ristrutturazione è necessaria perché il modello relazionale non permette di rappresentare in maniera diretta questo tipo di attributo. Quindi, un'entità viene legata ad un'altra entità tramite un'associazione **uno a molti**.

● Accorpamento di entità

L'accorpamento è l'operazione inversa del partizionamento, infatti due entità legate da un'associazione vengono accorpate in un'unica entità contenente gli attributi di entrambi. Un effetto collaterale di questa ristrutturazione la possibile presenza di *valori nulli*. Gli accorpamenti si effettuano su associazioni di tipo **uno a uno mai su relazioni molti a molti**.



Esempio di accorpamento di entità

4. Scelta degli identificatori principali

Questa scelta essenziale nelle traduzioni verso il modello relazionale virgola in modo da individuare una “**chiave primaria**” sulla quale vengono costruite delle strutture ausiliarie, dette indici, per la raccolta dei dati. I criteri di decisione sono i seguenti:

- Gli attributi con valori nulli non possono costituire identificatori principali.
- Un identificatore composto da uno o da pochi attributi è da preferire a identificatori costituiti da molti attributi. Questo infatti garantisce che gli indici siano di dimensioni ridotte, risparmiando memoria.
- Un identificatore interno con pochi attributi è da preferire ad un identificatore esterno che coinvolge diverse entità.
- Un identificatore che viene utilizzato da molte operazioni per accedere alle occorrenze di un'entità è da preferire rispetto agli altri.

Se nessuno degli identificatori soddisfa questi requisiti, è possibile introdurre un attributo che conterrà **valori specifici**, detti **codici**, generati per identificare le occorrenze dell'entità. E' consigliabile tenere traccia in questa fase anche di identificatori non selezionati che definiscono gli **indici secondari**.

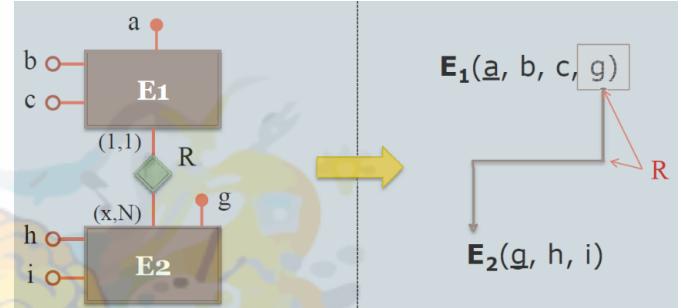
3.5 Traduzione verso il modello relazionale

La seconda fase della progettazione logica corrisponde alla **traduzione** che si realizza applicando allo schema concettuale ristrutturato un insieme di regole di traduzione. Ogni regola si applica ad un costrutto del modello concettuale ER e produce una o più strutture del modello relazionale.

Modello uno a molti

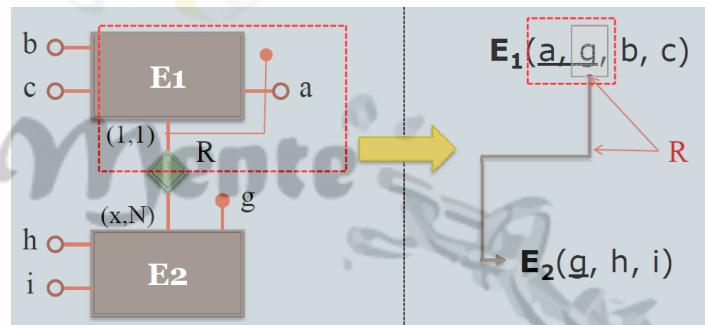
1.0 Modello uno a molti (con partecipazione obbligatoria)

L'entità che possiede cardinalità massima pari a 1 possiede anche gli identificatori delle altre entità coinvolte nell'associazione ma non come chiave.



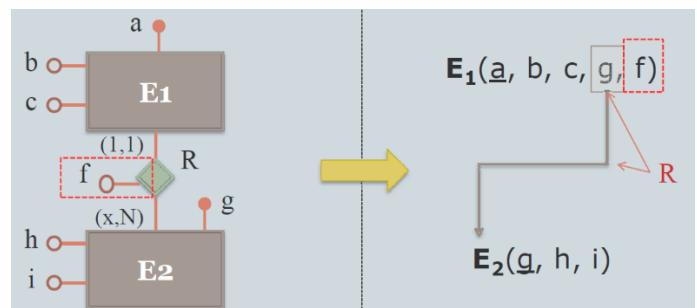
1.1 Modello uno a molti (identificatore esterno)

L'entità con l'identificatore esterno e cardinalità massima pari a 1 possiede anche gli identificatori delle altre entità coinvolte nell'associazione come chiave.



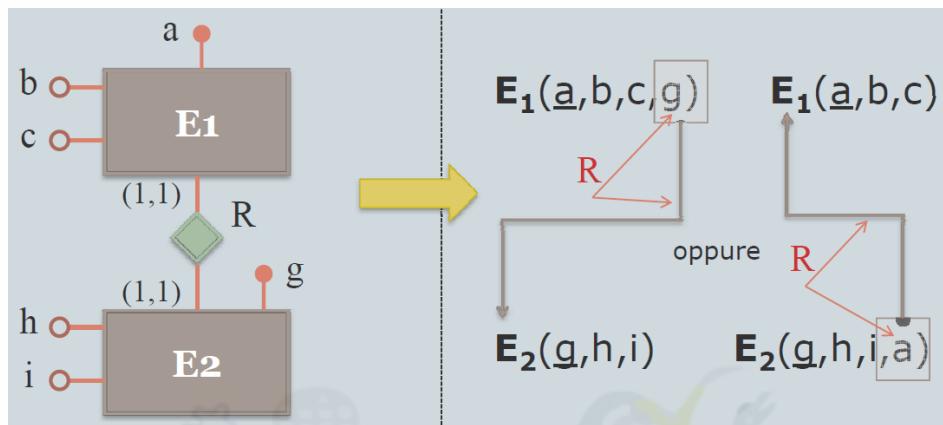
1.2 Modello uno a molti (con attributo sulla relazione)

L'entità con cardinalità massima pari a 1 possiede, oltre agli identificatori delle altre entità coinvolte, anche gli attributi situati sulla/e associazione/i coinvolte ma non come chiave.

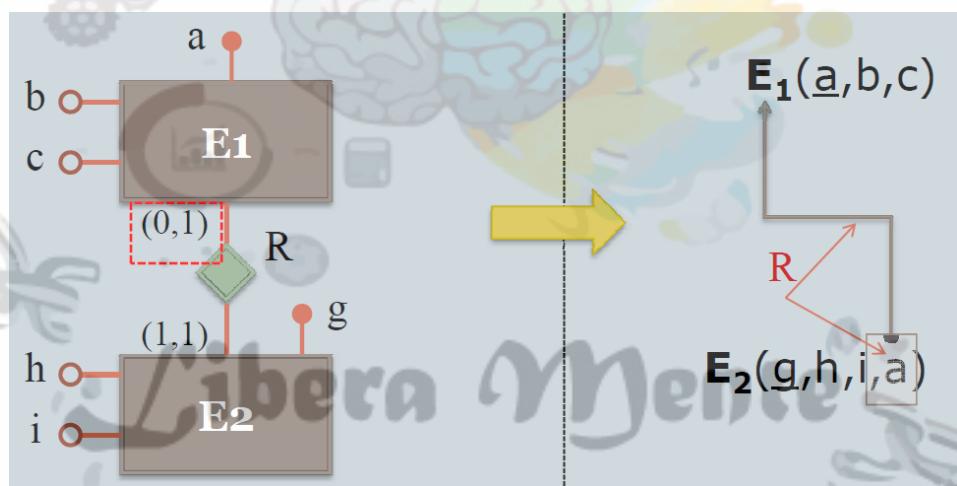


Modello uno a uno

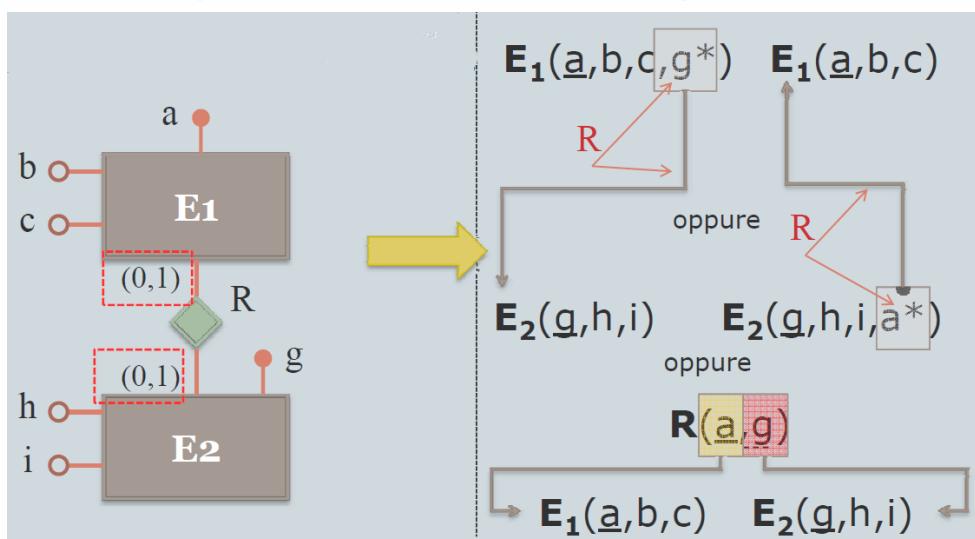
2.0 Relazione uno a uno



2.1 Relazione uno a uno (con cardinalità minima = 0)

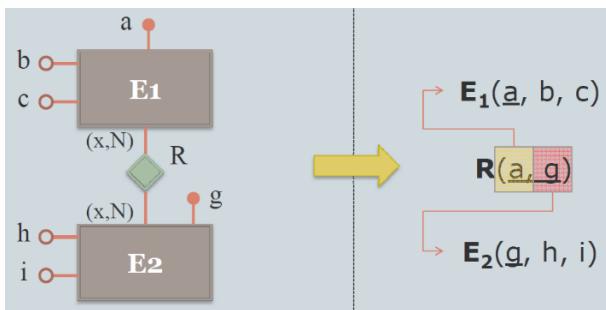


2.2 Relazione uno a uno (con due cardinalità minima = 0)

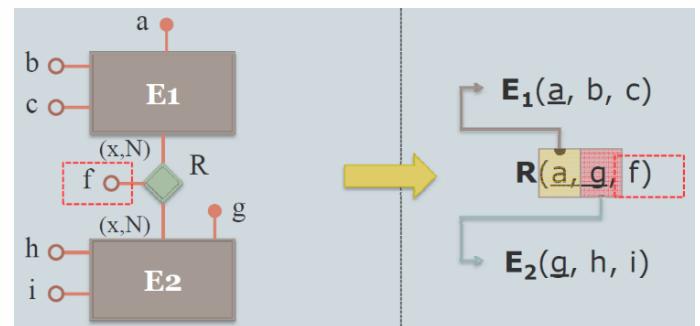


Modello molti a molti

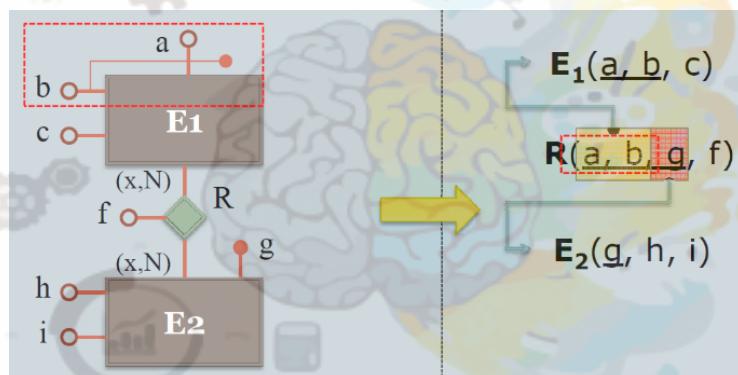
3.0 Relazione molti a molti



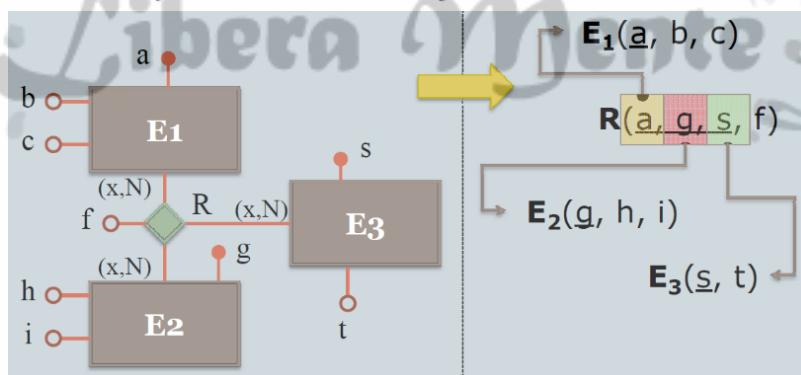
3.1 Relazione molti a molti (con attributi)



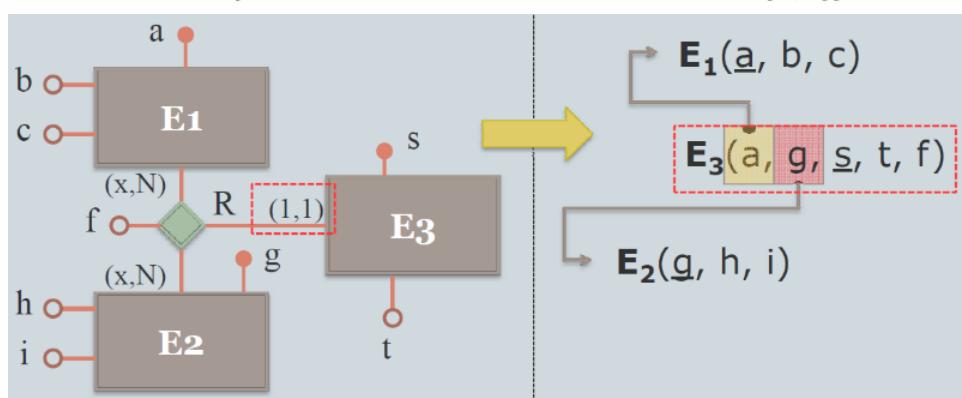
3.2 Relazione molti a molti (con identificatori e con più attributi)



3.3 Relazione molti a molti (relazione ternaria)



3.4 Relazione molti a molti (relazione ternaria con cardinalità (1,1))



Modello relazionale

4.1 Struttura modello relazionale

La maggior parte dei sistemi di basi dati oggi sul mercato si fonda sul **modello relazionale** che si basa su due concetti:

- **Relazione**: proviene dalla matematica, in particolare dalla teoria degli insiemi. Il termine relazione viene utilizzato in 3 circostanze:
 1. Matematica;
 2. secondo la definizione del modello relazionale;
 3. come traduzione di relationship e quindi con riferimento al modello concettuale.
- **Tabella**: concetto semplice e intuitivo.

4.1.1 Relazioni e tabelle

Una **relazione matematica** di due insiemi è definita come sottoinsieme del **prodotto cartesiano** di $D_1 \times D_2$, (l'insieme delle coppie ordinate (v_1, v_2) , tali che v_1 è un elemento di D_1 e v_2 è un elemento di D_2) questo perché gli insiemi, e quindi le relazioni, possono essere raggruppati in forma **tabellare**. Ma non tutte le tabelle esprimono una relazione.

Poiché le basi di dati devono essere memorizzate in sistemi di calcolo di dimensioni finite, le relazioni sono finite su domini eventualmente infiniti.

Il numero n di insiemi coinvolti viene detto “**grado della relazione**”. Il numero di elementi della relazione viene detto “**cardinalità della relazione**”. Ogni coppia è detta **n-upla**; scambiando l'ordine delle righe la rappresentazione è la stessa.

Una **relazione matematica** è un insieme di **n-uple** ordinate, in particolare:

- non c'è ordinamento fra le **n-uple**;
- le **n-uple** sono tutte diverse tra loro;
- ciascuna **n-upla** è ordinata: l'i-esimo valore proviene dall'i-esimo dominio.

Ad ogni colonna viene dato un nome detto **attributo** che descrive il “**ruolo**” giocato dal dominio stesso; in questo caso anche l'ordine delle colonne non è più importante. Nella rappresentazione tabellare, utilizziamo gli attributi come **intestazioni** quindi tali attributi devono essere diversi l'uno dall'altro. In questo caso, in una “**relazione con attributi**”, ogni elemento della rappresentazione è detto **tupla**.

Nella *relazione matematica* abbiamo *n-uple* i cui elementi sono individuati per posizione, mentre nelle *tuple* gli elementi sono individuati per mezzo degli attributi, cioè con una tecnica non posizionale.

Il **modello relazionale** è basato su **valori**, quindi rappresenta solo ciò che è rilevante: non vi è alcun legame tra livello logico e fisico (basato su puntatori e record) e ciò ha influenzato il successo del **modello relazionale**.

Una **Tabella** rappresenta una relazione se:

- i valori di ogni colonna sono fra loro omogenei;
- le righe sono diverse fra loro;
- le intestazioni delle colonne sono diverse tra loro.

Una **tabella** che rappresenta una relazione:

- l'ordinamento tra le righe è irrilevante;
- l'ordinamento tra le colonne è irrilevante.

4.2 Relazioni e basi di dati

- Uno schema di relazioni è costituito da un simbolo e un insieme di attributi
Esempio: Studente (Matricola, Nome)
- Uno schema di base dati è un insieme di queste relazioni;
- Un'istanza di relazione è un insieme di *tuple*;
- Una istanza di basi di dati è un insieme di relazioni dove ognuna è un'istanza di relazione.

Esempio finale: $R = \{ STUDENTI(Matricola, Nome, Cognome), ESAMI(Studente, Voto, Corso) \}$

4.3 Informazione incompleta e valori nulli

Le informazioni devono essere rappresentate per mezzo di *tuple*, in particolare in ogni relazione possiamo rappresentare solo *tuple* corrispondenti allo schema della relazione stessa.

Per rappresentare gli attributi i cui valori sono sconosciuti o non esistono per un valore nella *tupla* si utilizza il **valore nullo (NULL)**, che denota l'assenza di informazione. Il *valore nullo* deve essere usato con attenzione perché può generare dubbi sull'identità delle *tuple*.

Il valore nullo può assumere 3 caratteristiche:

1. Valore **sconosciuto**.
2. Il valore può essere **inesistente**.
3. Il valore inesistente oppure sconosciuto viene di solito chiamato **senza informazione**, perché non ci dice assolutamente niente: il valore può esistere o non esistere, e se esiste non sappiamo quale sia.

4.4 Vincoli di integrità

I dati vengono controllati dal *DBMS* e dall'applicazione, ma in genere le regole sui dati vengono controllate solo dal *DBMS*. Queste regole prendono il nome di **vincolo di integrità**, che devono essere rispettate da tutte le righe (*tuple*) di una tabella. Tra questi vincoli vi sono quelli per la definizione delle **chiavi**, tra cui la chiave primaria, e quelli che impongono delle regole sui valori assunti da due o più colonne.

Ogni vincolo può essere visto come un predicato che associa ad ogni istanza il valore *vero* o *falso*. Se il predicato assume il valore *vero*, allora diciamo che l'istanza soddisfa il vincolo.

Abbiamo diversi tipi di vincoli:

- **Vincolo intrarelazionale:** è un vincolo che definisce una singola relazione, ad esempio se ho una tabella "STUDENTE" ed inserisco una matricola, devo assicurarmi che nessun altro ha la stessa matricola scorrendo tutta la tabella.
 - **vincolo di tupla:** è un vincolo che può essere valutato su ciascuna *tupla* indipendentemente dalle altre; ad esempio se devo inserire una *lode*, vado a guardare semplicemente la *tupla* *voto*. La sintassi permette di definire espressioni booleane (AND, OR, NOT) che confrontano con gli operatori di uguaglianza, disuguaglianza e ordinamento, valori di attributo o espressioni aritmetiche su valori di attributo.
 - **vincolo di chiave:** sono i più importanti vincoli *intrarelazionali*, alla base del modello. Ad esempio, matricola non esistono due studenti con lo stesso numero di matricola.
- **Vincolo interrelazionale:** è un vincolo che coinvolge due o più relazioni. Per vedere se una regola è soddisfatta (all'interno di una tabella) bisogna andare ad analizzare un'altra tabella.

4.4.1 Chiavi

I **vincoli di chiave** sono i più importanti del modello relazionale. Una **chiave** è un insieme di attributi che identificano univocamente una *tupla* di una relazione. Ciascuna relazione e ciascuno schema di relazione ha sempre una chiave: in ogni relazione deve esistere una ed una sola **chiave primaria (primary key)**, ovvero una chiave che non contiene valori nulli (NULL).

In una tabella possono esserci molte chiavi ma soltanto una è anche una *chiave primaria*. Tutte le altre chiavi sono dette **chiavi secondarie (foreign key)**.

- Un insieme di attributi K si dice “**superchiave**” per la relazione r se r non contiene due *tuple* t_1 e t_2 tali che $t_1[k] = t_2[k]$. Un insieme di attributi è una **superchiave** se e solo se non contiene *tuple* duplicate al suo interno.
- Un insieme di attributi si dice chiave se è una “**superchiave minimale**”, ossia se non contiene altre *superchiavi* al suo interno.

4.4.2 Chiavi e valori nulli

Bisogna evitare di creare valori nulli sulle chiavi o sarebbe impossibile distinguere due righe di una tabella. Gli attributi che costituiscono la *chiave primaria* vengono spesso evidenziate attraverso la sottolineatura.

4.4.3 Vincoli di integrità referenziali

Un **vincolo di integrità referenziale** è la classe più importante tra i *vincoli interrelazionali*. Coinvolgono le relazioni esistenti tra le tabelle di base dello schema logico. Questo tipo di vincolo può essere applicato sia a una relazione uno a molti, dichiarando esplicitamente la *chiave esterna* nella tabella esterna (lato molti) sia a una relazione uno a uno, considerata come caso particolare di quella *uno a molti*, in cui una delle due *chiavi primarie* è anche la *chiave esterna*.

Sia un insieme di attributi X su R_1 e su R_2 , un vincolo referenziale è verificato se i valori di X di ogni *tupla* R_1 compaiono come valori di Y di R_2 , in genere Y è la *chiave primaria* di R_2 .

Questo vincolo si rappresenta come una freccia che va dalla tabella interna verso la tabella esterna.

Algebra e calcolo relazionale

5.1 Linguaggi per database

I linguaggi per la specifica delle operazioni (di *interrogazione* e *aggiornamento*) sui dati costituiscono una componente essenziale dei database.

- Un **aggiornamento** può essere visto come una funzione che, data un'istanza di un database, produce un'altra istanza sullo stesso schema.
- Un'**interrogazione (query)**, invece, è una funzione che, data un'istanza di un database, produce una relazione, su un dato schema. I linguaggi di interrogazione possono essere:
 - **procedurali**, come l'algebra relazionale, in cui le operazioni complesse vengono specificate descrivendo il procedimento da seguire per ottenere la soluzione ("come");
 - **dichiarativi** come il calcolo relazionale, in cui le espressioni descrivono le proprietà del risultato ("che cosa").

5.2 Algebra relazionale

L'**algebra relazionale** è un *linguaggio procedurale*, basato su concetti di tipo algebrico. E' costituito da un insieme di operatori, definiti su **relazioni** che producono ancora relazioni come risultati e tali relazioni possono essere composti. L'**algebra relazionale** produce tabelle **eliminando le tuple ripetute** al contrario di **SQL che le mantiene**. E' possibile costruire espressioni che coinvolgono più operatori, in modo da formulare interrogazioni anche complesse. I vari *operatori* si distinguono in:

- quell'insiemistici tradizionali: **unione**, **intersezione**, **differenza**;
- quelli più specifici: **ridenominazione**, **selezione**, **proiezione**;
- il più importante, quello di **join**, in varie forme, **join naturale**, **prodotto cartesiano** e **theta join**.

5.2.1 Operatori insiemistici

Le **relazioni** sono *insiemi*, per questo motivo possiamo definire su di essi gli operatori insiemistici di *unione*, *differenza* e *intersezione*. Una relazione è un insieme di **tuple omogenee**, definite sugli stessi attributi. Pertanto, gli operatori devono essere definiti su relazioni con attributi uguali.

1. l'**unione** di due relazioni r_1 e r_2 definite sullo stesso insieme di attributi X è ancora una relazione su X contenente le *tuple* che appartengono a r_1 oppure a r_2 , oppure a entrambe;
2. l'**intersezione** tra due relazioni su un insieme X è ancora una relazione su X contenente le *tuple* che appartengono sia a r_1 sia a r_2 ;
3. la **differenza** tra due relazioni su un insieme X è ancora una relazione su X contenente le *tuple* che appartengono a r_1 ma non a r_2 .

L'operazione di unione e intersezione è un'operazione commutativa mentre la differenza non lo è. L'unione e la differenza sono operatori fondamentali mentre l'intersezione è un operatore derivabile (dalla differenza).

5.2.2 Ridenominazione

La **ridenominazione** è un operatore unario ed ha come unico obiettivo quello di modificare i nomi degli attributi in modo da facilitare le operazioni insiemistiche che possono essere effettuate solo su attributi uguali. La ridenominazione "modifica il nome degli attributi" lasciando inalterato il contenuto delle relazioni: cambia solo l'intestazione, mentre il corpo rimane invariato. Il simbolo che viene utilizzato è ρ .

Esempio: $\rho_{Genitore \leftarrow Padre}(PATERNITA')$.

Padre	Figlio		Genitore	Figlio
Adamò	Abele	→	Adamò	Abele
Adamò	Caino		Adamò	Caino
Abramo	Isacco		Abramo	Isacco

5.2.3 Selezione

Gli operatori che permettono di manipolare le relazioni sono: la **selezione**, la **proiezione** e **join** (quest'ultimo con diverse varianti). La **selezione** e la **proiezione** sono operatori unari e producono come risultato una porzione dell'operando.

La **selezione** produce un sottoinsieme delle *n-uple*, su tutti gli attributi, mentre la **proiezione** dà un risultato in cui contribuiscono tutte le *tuples*, ma su un sottoinsieme degli attributi.

La **selezione** genera “decomposizioni orizzontali” e la **proiezione** “decomposizioni verticali”. Inoltre, la selezione può prevedere confronti fra attributi tramite connettivi logici: V, \wedge , \neg .

Il simbolo della selezione è : $\sigma_{Condizione} (Operando)$.

Esempio: $\sigma_{Stipendio > 50} (Impiegati)$.

Matricola	Cognome	Filiale	Stipendio		Matricola	Cognome	Filiale	Stipendio
7309	Rossi	Roma	55	→	7309	Rossi	Roma	55
5998	Neri	Milano	64		5998	Neri	Milano	64
9553	Milano	Milano	44					
5698	Neri	Napoli	64		5698	Neri	Napoli	64

5.2.4 Proiezione

La **proiezione** è anch'essa un operatore unario. Produce come risultato tante *tuple* quante l'operando, definite però solo su una parte degli attributi. Essendo le relazioni definite come insiemi, non possono comparire in esse più *tuple* uguali fra loro: i contributi uguali “**collassano**” in una sola *tupla*. In generale, possiamo dire che il risultato di una proiezione contiene al più tante *tuple* quante l'operando, ma può contenerne anche di meno. Esiste un legame fra i vincoli di chiave e le proiezioni:

- se X è una *superchiave* di R, allora $\pi_X(R)$ contiene esattamente tante n-uple quante R.

Il simbolo della proiezione è $\pi_{ListaAttributi} (Operando)$.

Esempio: $\pi_{Matricola, Cognome} (Operando)$

Matricola	Cognome	Filiale	Stipendio		Matricola	Cognome		
7309	Rossi	Roma	55	→	7309	Rossi		
5998	Neri	Milano	64		5998	Neri		
9553	Milano	Milano	44		9553	Milano		

5.2.5 Prodotto cartesiano

Il **prodotto cartesiano** è un'operazione binaria: contiene sempre un numero di *n-uple* pari al prodotto delle cardinalità degli operandi (le *n-uple* sono tutte combinabili).

Esempio: Impiegati × Reparti

Impiegati

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

Reparti

Codice	Capo
A	Mori
B	Bruni

Impiegati × Reparti

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

5.2.6 Join

Il **join** è l'operatore più interessante dell'algebra relazionale, in quanto permette di operare su dati contenuti in relazioni diverse. Esistono due versioni dell'operatore: la prima (**join naturale**) utile per riflessioni di tipo astratto e la seconda (**theta-join**) usata dal punto di vista pratico.

Join naturale

Il **join naturale** è un operatore binario che correla dati in relazioni diverse. Il risultato del *join* è costituito da una relazione sull'unione degli attributi degli operandi e le sue *tuple* sono ottenute combinando le *tuple* degli operandi con valori uguali sugli attributi comuni. Il grado della relazione ottenuta dal join è \leq della somma dei gradi dei due operandi, perché gli attributi omonimi degli operandi compaiono una sola volta nel risultato. Inoltre, il **join naturale** è **commutativo** e **associativo**. Il simbolo del **join** è \bowtie .

Quando ogni *tupla* di ciascuno degli operandi contribuisce ad almeno una *tupla* del risultato, il **join** si dice **completo** (in questo caso l'esempio 1 è un *join* completo).

Esempio 1: Impiegati \bowtie Reparti

Impiegati

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

Reparti

Reparto	Capo
A	Mori
B	Bruni

Impiegati \bowtie Reparti

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

E' possibile che nessuna *tupla* degli operandi sia combinabile, e allora si avrà un **join vuoto**.

Esempio 2: Impiegati \bowtie Reparti

Impiegati

Impiegato	Reparto
Rossi	A
Neri	B

Reparti

Reparto	Capo
D	Mori
C	Bruni

Impiegati \bowtie Reparti

Impiegato	Reparto	Capo

Quando alcune *tuple* degli operandi non contribuiscono al risultato, allora tali *tuple* verranno chiamate **dangling** e si avrà un **join incompleto** (*in questo caso l'esempio 3 è un join incompleto*).

Esempio 3: Impiegati \bowtie Reparti

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

Reparto	Capo
B	Mori
C	Bruni



Impiegati \bowtie Reparti

Impiegato	Reparto	Capo
Neri	B	Mori
Bianchi	B	Mori

Cardinalità del join

Il *join* di R1 e R2 contiene un numero di *n-uple* compreso fra 0 e il prodotto di |R1| e |R2|

- Se il *join* coinvolge una **chiave** di R2, allora il numero di *n-uple* è compreso fra 0 e |R1|;
- Se il *join* coinvolge una **chiave** di R2 e un **vincolo di integrità referenziale**, allora il numero di *n-uple* è pari a |R1|.

Join esterno e interno

Il **join esterno** (*outer join*) estende, con valori nulli, le *n-uple* che verrebbero tagliate fuori da un **join interno** (*inner join*). Esistono tre varianti dello *join esterno*:

- **Sinistro**: estende, con valori nulli se necessario, solo le *tuple* del primo operando;
- **Destro**: che estende solo le *tuple* del secondo operando;
- **Completo**: estende tutte le *tuple* di entrambi gli operandi;

Impiegati

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

Reparti

Reparto	Capo
B	Mori
C	Bruni



Impiegati \bowtie_{LEFT} Reparti

Impiegato	Reparto	Capo
Neri	B	Mori
Bianchi	B	Mori
Rossi	A	NULL

Impiegati

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

Reparti

Reparto	Capo
B	Mori
C	Bruni



Impiegati \bowtie_{RIGHT} Reparti

Impiegato	Reparto	Capo
Neri	B	Mori
Bianchi	B	Mori
NULL	C	Buni

Impiegati

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

Reparti

Reparto	Capo
B	Mori
C	Bruni



Impiegati \bowtie_{FULL} Reparti

Impiegato	Reparto	Capo
Neri	B	Mori
Bianchi	B	Mori
Rossi	A	NULL
NULL	C	Buni

Se due insiemi $X_1 = X_2$, l'operazione di join coincide con l'intersezione. Se $X_1 \neq X_2$ sono disgiunti l'operazione di join coincide con il prodotto cartesiano.

Theta-join

Il **theta-join** è un *operatore derivato* (cioè esprimibile per mezzo di altri operatori), come *prodotto cartesiano* seguito da una *selezione*: la **condizione F** è spesso una congiunzione (AND) di atomi di confronto $A_1 \theta A_2$ dove θ è uno degli operatori di confronto ($=, >, <, \dots$).

$$R_1 \bowtie_F R_2 = \sigma_F(R_1 \bowtie R_2)$$

Se l'operatore è sempre l'uguaglianza ($=$) allora si parla di **equi-join**.

Impiegati

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

Reparti

Codice	Capo
A	Mori
B	Bruni

Impiegati $\bowtie_{Reparto} = Codice$ Reparti

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



5.3 Equivalenza di espressioni

L'algebra relazionale permette di formulare **espressioni fra loro equivalenti**, cioè che producono lo stesso risultato. L'equivalenza di espressioni dell'algebra è importante nella fase di esecuzione delle query perché i DBMS cercano di eseguire espressioni equivalenti a quelle date, ma meno "costose". Per questo scopo vengono spesso utilizzate **trasformazioni di equivalenza**, cioè operazioni che sostituiscono un'espressione con un'altra a essa equivalente: tali trasformazioni riducono le dimensioni dei risultati intermedi.

5.4 Algebra con valori nulli

Un predicato può essere vero, falso oppure assumere un terzo nuovo valore di verità chiamato **unknown** (*sconosciuto*) rappresentato con il simbolo **U**. Un predicato assume questo valore quando almeno uno dei due termini del confronto assume il valore nullo. La condizione atomica è vera solo per valori non nulli. Per riferirsi ai valori nulli esistono due forme apposite:

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

not	
F	V
U	U
V	F

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

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

tabelle di verità

SQL

6.1 Introduzione

SQL (*Structured Query Language*) è il linguaggio per i database relazionali. **SQL** contiene al suo interno sia le funzionalità di un **DDL** (con un insieme di comandi per la definizione dello schema di un database relazionale), sia quelle di un **DML** (con un insieme di comandi per la modifica e l'interrogazione dell'istanza di un database).

SQL è un linguaggio di tipo **dichiarativo**: specifica le proprietà dei dati che devono essere estratti da un database, ma NON i dettagli del processo di estrazione.

6.2 I domini elementari

SQL mette a disposizione alcune famiglie di dominio elementari.

- **Caratteri**: il dominio `character` permette di rappresentare singoli caratteri oppure stringhe. La lunghezza delle stringhe di caratteri può essere fissa (`char`) o variabile (`varchar`); per le stringhe di lunghezza variabile si indica la lunghezza massima. Se viene inserita una stringa con meno caratteri rispetto alla dimensione fissata, non viene dato errore perché il sistema inserisce automaticamente spazi bianchi.
- **Tipi numerici esatti**: questa famiglia contiene domini che permettono di rappresentare valori esatti, interi o in virgola fissa. SQL mette a disposizione quattro diversi *tipi numerici esatti*:
 - `numeric`: si rappresenta valori compresi fra -99.9999 e +99.9999;
 - `decimal`: la differenza tra `numeric` e `decimal` consiste nel fatto che la precisione per il dominio “`numeric`” rappresenta un valore esatto, mentre per il dominio “`decimal`” rappresenta un valore minimo.
 - `integer`: rappresentazione su 4 byte;
 - `smallint`: intero rappresentato su 2 byte;
- **Tipi numerici approssimati**: i valori reali approssimati sono `float`, `real`, `double`.
- **Istanti temporali**: descrive informazioni temporali
 - `date` {year, month, day}
 - `time` {hour, minute, second}
 - `timestamp`: date time
- **Intervalli di tempo**: questi domini permettono di rappresentare intervalli di tempo, come per esempio la durata di un evento (`interval PrimaUnitàDiTempo [to UltimaUnitàDiTempo]`). *PrimaUnitàDiTempo* e *UltimaUnitàDiTempo* definiscono le unità di misura che devono essere usate, dalla più precisa alla meno precisa.
- **Boolean**: permette di rappresentare singoli valori, *true* o *false*.
- **BLOB e CLOB**: permettono, entrambi, di rappresentare oggetti di grandi dimensioni, costituiti da una sequenza di valori binari o di caratteri.

6.2.1 Definizione delle tabelle (`create table`)

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

Ogni tabella viene definita associandole un nome ed elencando gli attributi. Ogni attributo ha un nome, un dominio ed eventualmente un insieme di vincoli che devono essere rispettati dai valori dell'attributo. Inizialmente una tabella è vuota.

Esempio:

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

6.2.2 Definizione dei domini (`create domain`)

E' possibile creare nuovi domini tramite la primitiva `creative domain`. Un dominio è caratterizzato dal proprio nome, da un dominio elementare, da un eventuale valore di `default`, infine un insieme di vincoli che rappresentano un insieme di condizioni che devono essere rispettate dai valori del dominio. Questo metodo è utile per apportare modifiche velocemente, ma non è un costrutto di tipo.

Sintassi:

`create domain NomeDominio as TipoData [Vincolo]` → se non specifico il vincolo va messo 0

Esempio:

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

6.2.3 Specifica di valori di default

Il valore `default` è un termine che può essere inserito in corrispondenza di ogni dominio e attributo. Tramite il valore di `default`, l'attributo assume un valore quando viene inserita una riga nella tabella senza che sia specificato un valore per l'attributo stesso. Quando il valore di `default` non viene specificato si assume come default il `valore nullo`:

Esempio:

`NumeroFigli smallint default 0` → L'attributo `NumeroFigli` ammette come valore un numero intero e ha come valore di default 0.

6.2.4 Vincoli intrarelazionali

Il costrutto più potente per specificare vincoli generici, sia intrarelazionali che interrelazionali, è il costrutto `check` utilizzato per specificare un vincolo di tupla. I vincoli più semplici di tipo intrarelazionale sono:

1) Not null

Il `valore nullo` è un particolare valore che indica l'assenza di informazioni. SQL permette di utilizzare il vincolo `not null` che indica che il `valore nullo` non è ammesso come valore dell'attributo e quindi l'attributo in questione possiede delle informazioni. Se l'attributo ha un valore di `default` diverso dal `valore nullo`, allora l'inserimento avviene anche senza specificare un determinato valore per l'attributo, in quanto questo possiede già delle informazioni di `default`.

Esempio:

`cognome varchar(20) not null`

2) Unique

Il vincolo **unique** si applica a un attributo o a un insieme di attributi di una tabella e indica che i valori dell'attributo siano una (super)chiave, cioè non possono esistere righe di una tabella con valori uguali; viene fatta eccezione per il *valore nullo*, il quale può comparire su diverse righe poiché si assume che i *valori nulli* siano tutti diversi tra loro.

Esempio:

- Matricola character (6) **unique** → specificato su un singolo attributo;
- **unique** (Cognome, nome) → specificato su un insieme di attributi.

Esiste un caso particolare nell'uso del vincolo unique:

Nome CHAR(20) NOT NULL,
Cognome CHAR(20) NOT NULL,
UNIQUE (Cognome, Nome),

In questo caso si impone che non ci siano due righe che abbiano uguali sia il nome sia cognome.

Nome CHAR(20) NOT NULL UNIQUE,
Cognome CHAR(20) NOT NULL UNIQUE,

In questo caso si ha una violazione se nelle righe compaiono più di una volta o lo stesso cognome o lo stesso nome.

3) Primary key

Per specificare la **chiave primaria** per ogni relazione, SQL permette di utilizzare il vincolo **primary key** una sola volta per ogni tabella.

Il vincolo **primary key** può essere definito su un singolo attributo, oppure essere definito elencando gli attributi che costituiscono l'identificatore. Gli attributi che fanno parte della chiave primaria non possono assumere il *valore nullo*.

Esempio 1)

```
create table Dipartimento
(
    Nome varchar(20) primary key,
    Città varchar(20)
)
```

Esempio 2)

```
create table Dipartimento
(
    Nome varchar(20),
    Città varchar(20),
    primary key (Nome)
)
```

Esempio 3)

```
create table Impiegato
(
    Nome varchar(20),
    Cognome varchar(20),
    primary key (Nome, Cognome)
)
```

Significa che possono esistere *impiegati* con lo stesso *nome* e *cognome*, ma NON due persone con lo stesso nome e cognome.

In sintesi, i **vincoli intrarelazionali** possono essere combinati tra loro: *not null + unique* rappresenta una sorta di chiave primaria.

6.2.5 Vincoli interrelazionali

In SQL esiste il vincolo **foreign key** (*chiave esterna*) che crea un legame tra l'attributo della tabella corrente (*interna*) e l'attributo di un'altra tabella (*esterna*). Il vincolo viene definito sulla tabella interna. Spesso è richiesto che l'attributo della tabella esterna sia identificato dal comando **unique**. Il vincolo può essere definito in 2 modi:

1. Se c'è un solo attributo coinvolto, si può usare il costrutto sintattico **references**;
2. Se abbiamo un insieme di attributi, useremo il costrutto **foreign key** che elenca gli attributi della tabella coinvolti nel legame, cui segue il costrutto **references** con gli attributi della tabella esterna

1) **create table** Impiegato

```
(  
    Nome varchar(20),  
    Cognome varchar(20),  
    Dipart varchar(15) references Dipartimento(NomeDip)  
)
```

2) **foreign key**(Nome, Cognome) **references** Anagrafica(Nome, Cognome)

Non è possibile violare un vincolo ma ci sono operazioni che impattano sul vincolo a livello di tabella: sulla *tabella interna* impattano le operazioni di **inserimento** e **modifica**; sulla *tabella esterna* impattano le operazioni di **modifica** e **cancellazione**.

Ciò che fa il sistema è di evitare questo tipo di operazioni sulla tabella interna; sulla tabella esterna dobbiamo distinguere le due operazioni:

1) **Per la modifica:**

- **cascade**: il nuovo valore dell'attributo della tabella esterna viene riportato su tutte le corrispondenti righe della tabella interna;
- **set default**: all'attributo viene assegnato il valore di *default* al posto del valore modificato nella tabella esterna;
- **set null**: in questo caso viene assegnato il valore null;
- **no action/restrict**: la modifica non viene consentita.

2) **Per la cancellazione:**

- **cascade**: tutte le righe della tabella interna corrispondenti alla riga cancellata vengono cancellate;
- **set default**: viene assegnato il valore di *default*;
- **set null**: all'attributo viene assegnato null al posto del valore cancellato nella tabella esterna;
- **no action/restrict**: la cancellazione non viene consentita.

Il valore di default su queste politiche è la "no action".

Esempio:

```
create table Impiegato  
(
```

```
    Nome varchar(20),  
    Cognome varchar(20),  
    Dipart varchar(15)  
        references Dipartimento(NomeDip)  
        on delete set null
```

```

        on update cascade,
Ufficio numeric(3),
primary key(Matricola),
unique (Cognome, Nome)
)

```

6.2.6 Modifica degli schemi

SQL fornisce comandi che permettono di modificare le tabelle viste precedentemente:

- ❖ **alter**: permette di modificare domini e schemi di tabelle. È possibile aggiungere e rimuovere vincoli e modificare i valori di default; è possibile aggiungere ed eliminare attributi e vincoli.

- alter domain *NomeDominio* (set default *ValoreDefault* | drop default | add constraint *DefVincolo* | drop constraint *NomeVincolo*)
- alter table *NomeTabella* (
 alter column *NomeAttributo* (set default *NuovoDefault* | drop default
 add constraint *DefVincolo* |
 drop constraint *NomeVincolo* |
 add column *DefAttributo* |
 drop column *NomeAttributo*)
)

Esempio:

```
alter table Dipartimento add column NroUff numeric(4)
```

- ❖ **drop**: permette di rimuovere dei componenti

- drop(schema | domain | table | view | assertion) *NomeElemento* [restrict | cascade]

Restrict specifica che il comando non deve essere eseguito in presenza di oggetti non vuoti; cascade, invece, implica che tutti gli oggetti devono essere rimossi.

6.2.7 Cataloghi relazionali

Tutti i DBMS relazionali gestiscono il proprio “dizionario dei dati” mediante una struttura relazionale. Il database contiene quindi due tipi di tabelle: quelle che contengono i dati e quelle che gestiscono i **metadati** (dati che descrivono dati). Questo secondo insieme di tabelle costituisce il **catalogo** del database. Lo standard SQL prevede due livelli:

1. DEFINITION-SCHEMA, costituito da un insieme di tabelle che contengono la descrizione di tutte le strutture del database;
2. INFORMATION-SCHEMA, un insieme di viste che costituiscono un’interfaccia verso il dizionario dei dati. Le viste principali sono: *tables*, *views*, *columns*, *domains*...

6.3 Query in SQL

La parte di SQL dedicata alla formulazione di interrogazioni fa parte del DML. SQL esprime le query in modo **dichiarativo**, ovvero si specifica l'obiettivo della query e non il modo in cui ottenerlo. Questo metodo si contrappone a *linguaggi di interrogazione procedurali*, come l'algebra relazionale.

Ordine di esecuzione delle clausole:

From → Where → Group by → Having → Order by → Select

6.3.1 Query semplici

Una query produce una relazione (tabella). Le operazioni di interrogazione in SQL vengono specificate per mezzo dell'istruzione **select**. Le tre parti di cui si compone un'istruzione **select** sono spesso chiamate clausola **select**, clausola **from** e clausola **where**.

Clausola select. La clausola **select** produce gli elementi della tabella risultato. Come argomento può anche comparire il carattere speciale *****, che rappresenta la selezione di tutti gli attributi delle tabelle elencate nella **from**. La **select** corrisponde all'operazione di *proiezione* in algebra relazionale. Il risultato di una **select** non è una relazione, bensì una tabella, poiché ci possono essere righe uguali.

Clausola from. La clausola **from** ha come argomento l'insieme di tabelle alle quali si vuole accedere. Sul prodotto cartesiano delle tabelle elencate verranno applicate le condizioni contenute nella clausola **where**. La **from** corrisponde all'operazione di *Prodotto cartesiano* in algebra relazionale

Clausola where. La clausola **where** ammette come argomento un'espressione booleana costruita combinando predicati semplici con gli operatori **and**, **or** e **not**. Se necessario esprime una query che richiede l'uso di più operatori ordinati attraverso l'uso delle parentesi. La **clausola where** corrisponde alla *selezione* in algebra relazionale.

L'operatore **like** è usato nel confronto tra stringhe e usa due caratteri speciali: **_** indica qualsiasi carattere e **%** indica una stringa, anche vuota. Like è case sensitive.

6.3.2 Gestione dei valori nulli

Un valore nullo in un attributo può significare che un certo attributo non è applicabile, o che il valore è applicabile ma non è conosciuto. Per selezionare i termini con valori nulli SQL fornisce il predicato **is null**, mentre il predicato **is not null** è la sua negazione.

Il predicato **is null** restituisce sempre il valore *vero* o il valore *falso*, e mai il valore *unknown*.

6.3.3 Duplicati

Una differenza tra SQL e l'algebra relazionale data dalla gestione dei duplicati. In SQL i duplicati vengono mantenuti a differenza dell'algebra relazionale. Per emulare il comportamento dell'algebra relazionale, sarebbe necessario effettuare l'eliminazione dei duplicati tutte le volte in cui si eseguono operazioni di proiezione. L'operazione di rimozione di duplicati è però molto costosa e spesso non necessaria.

L'eliminazione dei duplicati è specificata da **distinct**, posta subito dopo la **select**. La parola **all** indica che si intendono mantenere tutti i duplicati ma può essere omessa poiché di default vengono mantenuti i duplicati.

6.3.4 Join interni ed esterni

La condizione di **join** viene usata all'interno della clausola **from** o all'interno della clausola **where** dipende se utilizzeremo un **join esplicito** o **implicito**. Abbiamo diversi tipi di **join**:

- 1) **inner** – interno, valore di default;
- 2) **right outer** – mantiene le righe della tabella di destra;
- 3) **left outer** – mantiene le righe della tabella di sinistra;
- 4) **full outer** – mantiene le righe di entrambe le tabelle.

Con il **join interno** le righe che vengono coinvolte nel join sono un sottoinsieme delle righe di ciascuna tabella (può capitare che non ci siano corrispondenze sulle righe).

Esempio join esplicito: la condizione viene espressa nella clausola **from**

```
select *  
from Impiegato as I join Dipartimento as D on (I.Dipartimento = D.Nome)
```

Esempio join implicito: la condizione viene espressa nella clausola **where**

```
select *  
from Impiegato as I, Dipartimento as D  
where I.Dipartimento = D.Nome
```

6.3.5 Uso di variabili

Utilizzando gli **alias** è possibile fare accesso più volte alla stessa tabella. Tutte le volte che si introduce un alias per una tabella si dichiara in effetti una variabile che rappresenta le righe della tabella di cui è alias.

Attraverso la ridenominazione **as** nel **from** è possibile richiamare più volte la stessa tabella senza cadere in ambiguità.

6.3.6 Ordinamento (**order by**)

Il risultato di una query è una relazione non ordinata. SQL permette di specificare un ordinamento delle righe del risultato di una query tramite la clausola **order by**:

```
order by Attributi [asc | desc].
```

L'ordinamento agisce prima della **select**. Le righe vengono ordinate in base al primo attributo nell'elenco. Per righe che hanno lo stesso valore del primo attributo, si considerano i valori degli attributi successivi, in sequenza. L'ordine su ciascun attributo può essere ascendente o discendente, a seconda che si sia usato il qualificatore **asc** o **desc**.

Esempio:

```
select *  
from Automobile  
order by Marca desc, Modello
```

6.3.7 Operatori aggregati

Gli **operatori aggregati** costituiscono una delle più importanti estensioni di SQL rispetto all'algebra relazionale. In algebra relazione tutte le condizioni vengono valutate su una tupla alla volta. Lo standard SQL prevede 5 operatori aggregati: **count**, **sum**, **max**, **min** e **avg**.

Count

1) Select **count (*)**

From Impiegato → l'operazione restituisce il numero di righe della tabella Impiegato.

2) Select **count (distinct nome)**

From Impiegato → l'operazione restituisce il numero di valori diversi dall'attributo nome.

3) Select **count (all nome)**

From Impiegato → l'operazione restituisce il numero di valori che non sono null dall'attributo nome.

Inoltre, **distinct** elimina i duplicati mentre **all** trascura solo i valori nulli.

Gli altri 4 operatori aggregati ammettono un attributo o un'espressione come argomento:

(sum | max | min | avg) ([distinct | all] AttrEspr)

Le funzioni **sum** e **avg** si applicano solo a valori numerici; le funzioni **max** e **min** si applicano a tutti i tipi che permettono un ordinamento.

6.3.8 Interrogazioni con raggruppamento (group by)

Per poter applicare l'operatore aggregato a sottoinsiemi di righe, SQL mette a disposizione la clausola **group by**, che permette di specificare come dividere le tabelle in sottoinsiemi. La clausola ammette come argomento un insieme di attributi e la query raggrupperà le righe che possiedono gli stessi valori per questo insieme di attributi. La **group by** agisce dopo la **from** e dopo la **where**. Dopo che le righe sono state raggruppate in sottoinsiemi, l'operatore aggregato viene applicato separatamente ad ogni sottoinsieme.

Esempio:

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

Dipartimento	Stipendio
Amministrazione	45
Amministrazione	40
Amministrazione	40
Produzione	36
Produzione	40
Distribuzione	45
Direzione	80
Direzione	75

Query finale

Dipartimento	Stipendio
Amministrazione	125
Produzione	82
Distribuzione	45
Direzione	153

6.3.9 Clausola having

La clausola **having** scrive le condizioni che si devono applicare dopo la **group by**. Ogni sottoinsieme di righe costruito dalla **group by** fa parte del risultato della query solo se viene soddisfatta la condizione di **having**. Viene utilizzata la **having** anziché la **where** solo quando compaiono come prediciati gli operatori aggregati.

Esempio:

```
select Dipart  
from Impiegato  
where Ufficio = 20  
group by Dipart  
having avg(Stipendio) > 25
```

oppure

```
select Dipart, sum(Stipendio)  
from Impiegato  
where Ufficio = 20  
group by Dipart  
having sum(Stipendio) >= 20
```

6.3.10 Query di tipo insiemistico

SQL mette a disposizione anche degli **operatori insiemistici**, simili a quelli nell'algebra relazionale. Gli operatori disponibili sono 3: union, intersect ed except. Tali operatori operano sul risultato di una select ed eseguono sempre un'eliminazione dei duplicati se non si esplicita la parola chiave all. E' Importante che gli attributi siano in pari numero e che abbiano domini compatibili, ovvero stesso numero di colonne. Se gli attributi hanno un nome diverso, il risultato usa nomi del primo operando. Tali operatori possono essere sostituiti dalle **query nidificate**.

Esempio:

```
select Nome  
from Impiegato  
union  
select Cognome  
from Impiegato
```

→ estrae i nomi e cognomi degli impiegati

6.4 Query nidificate

In SQL è possibile confrontare un valore con il risultato di una select. Si confronta un valore (risultato di un'espressione) con il risultato dell'esecuzione di una query. L'interrogazione è definita direttamente nel predicato della clausola where (nidificata) e la nidificazione nella having non è ammessa.

Abbiamo due parole chiavi:

- 1) **any**: la riga soddisfa la condizione se risulta vero il confronto tra il valore dell'attributo per la riga e almeno uno degli elementi restituiti dall'interrogazione.
- 2) **all**: la riga soddisfa la condizione solo se tutti gli elementi restituiti dall'interrogazione 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.

Esempio:

```
select *  
from Impiegato  
where Dipart = any(select Nome  
from Dipart  
where citta = 'Firenze')
```

→ **query interna**

La query interna seleziona tutti i dipartimenti che hanno come città Firenze e la query esterna mi seleziona almeno uno degli impiegati selezionati dalla query interna. Quindi ciò che vado a fare è selezionare tutti gli impiegati che lavorano in un dipartimento che si trova a Firenze.

SQL permette, inoltre, di rappresentare condizioni di appartenenza o di esclusione rispetto a un insieme, mediante due appositi operatori: **in** e **not in**; i risultati sono del tutto identici agli operatori = any e <> all. Gli operatori in e not in sono molto utili nelle query nidificate.

Esempio:

```
SELECT CodImp, Sede  
FROM Imp  
WHERE Sede IN ('S02', 'S03')
```

Lo stesso risultato si ottiene scrivendo:

```
SELECT CodImp, Sede  
FROM Imp  
WHERE Sede = 'S02' OR Sede = 'S03'
```

6.4.1 (not) exists

Mediante **EXISTS** (SELECT * ...) è possibile verificare se il risultato di una *subquery* restituisce almeno una tupla. Facendo uso di **NOT EXISTS** il predicato è vero se la subquery non restituisce alcuna tupla. In entrambi i casi la cosa non è molto "interessante" in quanto il risultato della subquery è sempre lo stesso, ovvero non dipende dalla specifica tupla del blocco esterno.

Esempio:

```
select *  
from Persona P  
where exists (select *  
                  from Persona P1  
                  where P1.nome = P.Nome and  
                        P1.cognome = P.cognome  
                        P1.CodFiscale <> P.CodFiscale)
```

Questa query estrae le persone che hanno lo stesso nome e cognome, ma diverso codice fiscale. Si esegue prima la query esterna: la query interna viene eseguita tante volte quante sono le righe della tabella persona.

In fine diremo che:

- una subquery che restituisca al massimo un valore è detta **scalare**, e per essa si possono usare i soliti operatori di confronto;
- Le forme <op> ANY e <op> ALL si rendono necessarie quando la subquery può restituire più valori;
- Il quantificatore esistenziale **EXISTS** è soddisfatto quando il risultato della subquery non è vuoto (e **NOT EXISTS** quando è vuoto);
- Una subquery si dice **correlata** se riferenzia variabili definite in un blocco ad essa più esterno;
- In molti casi è possibile scrivere una query sia in forma piatta che in forma innestata.

6.5 Modifica dei dati in SQL

La parte di DML comprende i comandi per interrogare modificare il contenuto di un database. I comandi che permettono di modificare il database sono:

Inserimento

Il comando di inserimento di righe nel database presenta due sintassi diverse:

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

```
insert into Dipartimento (NomeDip, Città)
values ('Produzione', 'Torino')
```

- 2) La seconda forma permette di aggiungere degli insiemi di righe, estratti dal contenuto del database.

```
insert into ProdottiMilanesi
```

```
select Codice, Descrizione
from Prodotto
where LuogoProd = 'Milano')
```

Se non vengono specificati i valori in inserimento, vengono messi quelli di default o, in mancanza, il valore null.

Cancellazione

Il comando **delete** elimina righe dalle tabelle del database. Quando non viene specificata nessuna condizione la tabella viene svuotata, ma non cancellata. Ad esempio: **delete from Dipart**

Quando la condizione argomento nella clausola **where** viene specificata allora vengono rimosse solo le righe che soddisfano la condizione. Ad esempio: **delete from Dipart**

```
where NomeDip = 'Produzione'
```

Modifica

Il comando di **update** permette di aggiornare uno più attributi delle righe di una tabella che soddisfano una condizione. Se il comando non presenta la clausola **where** allora si effettua la modifica su tutte le righe.

```
update NomeTabella
set Attributo = <Espressione>
[where condizione]
```

6.6 Operatore **between**

L'operatore **BETWEEN** permette di esprimere condizioni di appartenenza a un intervallo.

Esempio:

```
SELECT Nome, Stipendio
FROM Imp
WHERE Stipendio BETWEEN 1300 AND 2000
```

La query restituisce *Nome* e *stipendio* degli impiegati che hanno uno stipendio compreso tra 1300 € e 2000 € (estremi inclusi).

SQL avanzato

7.1 Vincoli di integrità generici

SQL permette di specificare un certo insieme di vincoli sugli attributi e sulle tabelle. Per specificare vincoli più complessi, SQL-2 offre la clausola **check**: **check (Condizione)**.

La condizione deve essere sempre verificata affinché il database sia corretto. In questo modo è possibile specificare tutti i vincoli intrarelazionali.

Inoltre, con la clausola **check** si perde la possibilità di associare ai vincoli una politica di reazione alle violazioni: infatti, quando i vincoli sono espressi mediante costrutti predefiniti, il sistema li può riconoscere immediatamente e spesso può riuscire a gestirli in modo più efficiente.

Esempio:

```
create table Impiegato  
  (Matricola character(6)  
   check (Matricola is not null and  
           1 = (select count(*)  
                  from Impiegato I  
                 where Matricola = I.Matricola)),  
  
  Cognome character(20) check (Cognome is not null),  
  Nome character(20) check (Nome is not null and  
                           2 > (select count(*)  
                                 from Impiegato I  
                                where Nome = I.Nome  
                                  and Cognome = I.Cognome)),  
  
  Dipart character(15) check (Dipart in  
                            (select NomeDip  
                               From Dipartimento))  
)
```

7.1.1 Asserzioni

Grazie alla clausola **check** è possibile definire anche le **asserzioni**: rappresentano dei vincoli che non sono associati a un attributo o una tabella in particolare, bensì appartengono direttamente allo schema.

Mediane le **asserzioni** è possibile esprimere tutti i vincoli che abbiamo specificato nella definizione delle tabelle. Le asservizioni permettono inoltre di esprimere vincoli che coinvolgono più tabelle o che richiedono una tabella abbia una cardinalità minima. Le asservizioni possiedono un nome, tramite il quale possono essere eliminate dallo schema con l'istruzione **drop**.

La sintassi delle asservizioni è: **create assertion NomeAsserzione check (Condizione)**.

Esempio:

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

Ogni vincolo di integrità, definito tramite **check** o tramite **asserzione**, è associato a una politica di controllo che specifica se il vincolo è immediato o differito. I vincoli immediati sono verificati immediatamente da ogni modifica del database, mentre i vincoli differiti sono verificati solo al termine dell'esecuzione di una serie di operazioni.

Quando un vincolo immediato non è soddisfatto, l'operazione di modifica che ha causato la violazione è stata appena eseguita e il sistema può "disfarla"; questo modo di procedere è chiamato **rollback parziale**. Tutti i vincoli predefiniti (`not null`, `unique`, `primary key`, `foreign key`) sono per default verificati in modo immediato e la loro violazione causa un *rollback parziale*.

Quando si verifica una violazione di un vincolo differito non c'è modo di individuare l'operazione che ha causato la violazione, e perciò diventa necessario disfare l'intera sequenza di operazioni che costituiscono la transazione; in questo caso si esegue un **rollback**.

È possibile cambiare il tipo di controllo associato ai vincoli tramite i comandi `set constraints [NomeVincolo | all] immediate` e `set constraints [NomeVincolo | all] deferred`, che modificano la modalità di controllo dei vincoli nominati, o di tutti i vincoli se si usa l'opzione `all`.

7.1.2 Viste

Nel modello relazionale, per rappresentare gli stessi dati in maniera diversa viene utilizzata la tecnica delle **relazioni derivate**: relazione il cui contenuto è funzione del contenuto di altre relazioni, definito per mezzo di *query*. Possono esistere due tipi di relazioni derivate:

- **viste materializzate**: relazioni derivate e memorizzate nel database;
- **relazioni virtuali** (dette anche **viste**): relazioni definite per mezzo di funzioni che non sono memorizzate nel database, ma sono utilizzate nelle query.

Le viste materializzate hanno il vantaggio di essere immediatamente disponibili per le *query*, ma spesso sono molto onerose perché sono ridondanti, appesantiscono gli aggiornamenti e sono raramente supportate dai DBMS.

Le relazioni virtuali (o viste) devono essere ricalcolate per ogni *query* ma non presentano problemi di allineamento. Le viste vengono definite nei sistemi relazionali per mezzo di espressioni del linguaggio di interrogazione.

L'uso delle viste può essere vantaggioso per diversi motivi:

- ogni utente vede solo ciò che gli interessa e nel modo in cui gli interessa, senza essere distratto dal resto. Inoltre, può vedere solo ciò che gli è stato autorizzato;
- l'utilizzo di viste non influisce sull'efficienza delle interrogazioni;
- si può semplificare la scrittura di interrogazioni molto complesse.

Le **viste** vengono definite in SQL associando un nome e una lista di attributi al risultato dell'esecuzione di una query. Nell'interrogazione che definisce la vista possono comparire anche altre viste. Inoltre, l'interrogazione deve restituire un insieme di attributi compatibile con l'insieme di attributi della vista.

Esempio:

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

Le viste sono utilizzabili per scrivere query soprattutto quando è richiesto l'utilizzo combinato di operatori aggregati.

Esempio:

```
create view BudStip (Dip, TotStip) as  
select Dipart, sum(Stipendio)  
from Impiegato  
group by Dipart  
  
select Dip  
from BudStip  
where TotStip = (select max (TotStip)  
                  from BudStip)
```

Una vista può essere modificata solo se le modifiche possono essere applicate univocamente alle tabelle sottostanti, ovvero quando una sola riga di ciascuna tabella di base corrisponde a una riga della vista. Non tutti i sistemi permettono la modifica; quelli che lo permettono adottano comunque delle restrizioni. **check option** permette modifiche, ma solo a condizione che la tupla continui ad appartenere alla vista.

7.1.3 Viste ricorsive

La sintassi SQL-2 non permette di realizzare *viste ricorsive*, mentre SQL-3 offre supporto per le viste ricorsive, utilizzando una struttura di definizione simile al linguaggio *Datalog*. Una sorta di vista ricorsiva può essere realizzata con la clausola **with**:

```
WITH Antenati(Persona,Avo)  
AS ((SELECT Figlio, Genitore  
      FROM Genitori)  
     UNION ALL  
     (SELECT G.Figlio, A.Avo  
      FROM Genitori G, Antenati A  
      WHERE G.Genitore =  
            A.Persona))  
  
SELECT Avo  
FROM Antenati  
WHERE Persona = 'Anna'
```

→ subquery base
→ sempre UNION ALL!
→ subquery ricorsiva

7.2 Controllo dell'accesso

SQL prevede che ogni utente sia identificato in modo univoco dal sistema. Ciò può avvenire sia appoggiandosi al sistema operativo, sia con meccanismi propri del DBMS. Le risorse che il sistema protegge sono normalmente tabelle e viste. L'utente che crea la risorsa ne è il proprietario ed è autorizzato a compiere qualsiasi operazione.

7.2.1 Risorse e privilegi

Il sistema basa il controllo di accesso su un concetto di **privilegio**; gli utenti possiedono dei privilegi di accesso alle risorse del sistema e ogni privilegio ha dei parametri:

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

Quando una risorsa viene creata, il sistema concede automaticamente tutti i privilegi su tale risorsa al creatore. Esiste inoltre un utente predefinito, `_system`, che rappresenta il *database administrator*, il quale possiede tutti i privilegi su tutte le risorse. I privilegi disponibili sono i seguenti:

- `insert`: permette di inserire un nuovo oggetto (tabelle e viste);
- `update`: permette di aggiornare il valore di un oggetto (tabelle, viste, attributi);
- `delete`: permette di rimuovere oggetti (tabelle e viste);
- `select`: permette di leggere la risorsa, utilizzarla nell'ambito di una query (tabelle, viste e attributi);
- `references`: permette che venga fatto un riferimento a una risorsa nell'ambito della definizione dello schema di una tabella (tabelle, attributi);
- `usage`: permette che venga usata la risorsa, ma solo per risorse come i domini.

7.2.2 Comandi per concedere e revocare i privilegi

I privilegi vengono concessi revocati tramite le istruzioni **grant** e **revoke**.

Grant

Sintassi: `grant Privilegi on Risorsa to Utenti [with grant option]`.

Tale comando permette di concedere i *Privilegi* sulla *Risorsa* agli *Utenti*.

Per esempio: `grant select on Dipartimento to Stefano`.

La clausola “`with grant option`” specifica se deve essere concesso (a Stefano) anche il privilegio di propagare il privilegio di altri utenti. È possibile usare al posto dei *privilegi* la parola chiave `all privileges`, che identifica tutti i privilegi che l'utente può concedere sulla particolare risorsa.

Esempio:

`grant all privileges on Impiegato to Paolo, Riccardo`

Revoke

Sintassi: `revoke Privilegi on Risorsa from Utenti [restrict | cascade]`.

Tale comando fa invece l'inverso: sottrae a un utente privilegi che gli erano stati concessi. Il comando `revoke` può eliminare tutti i privilegi che erano stati concessi, o limitarsi a revocarne un sottoinsieme.

L'opzione “`restrict`” è il valore di default e specifica che il comando non deve essere eseguito qualora la revoca dei privilegi all'utente comporti qualche altra revoca di privilegi, come può capitare quando l'utente ha ricevuto i privilegi con la `grant option`.

Con l'opzione “`cascade`” tutti i privilegi che erano stati propagati vengono revocati a tutti: bisogna stare attenti ad una reazione a catena.

Normalizzazione

8.1 Introduzione

La **normalizzazione** agisce a livello logico: fornisce metodi per migliorare la qualità del progetto o del database, mediante la quale si eliminano le ridondanze dei dati al fine di evitare anomalie in seguito a operazioni di inserimento, cancellazione o modifica. La normalizzazione viene eseguita in varie fasi. Al termine di ciascuna fase il database si trova in uno degli stati di *normalizzazione*, dette "**forme normali**".

Una **forma normale** è una proprietà che deve essere soddisfatta dalle relazioni di uno schema. Tale proprietà fornisce un certo livello di "qualità" di uno schema. Quando una relazione non è normalizzata presenta ridondanze, non consente ricerche veloci e crea difficoltà durante le operazioni di aggiornamento.

8.2 Dipendenza funzionale

Per scoprire e rimuovere le anomalie in uno schema logico si devono individuare legami di tipo funzionale tra gli attributi, dette **dipendenze funzionali**.

La notazione per indicare una dipendenza funzionale è: $Y \rightarrow Z$ (Y determina Z).

Dato uno schema di relazione $R(x)$ e due sottoinsiemi di attributi Y e Z , diremo che in $R(x)$ esiste una dipendenza funzionale $Y \rightarrow Z$ se: per ogni coppia di tuple appartenenti a R , se le due tuple sono uguali su Y , sono necessariamente uguali in Z , per ogni istanza di R .

Inoltre, una DF è **banale** perché afferma una proprietà ovvia di una relazione (sempre soddisfatta), mentre una DF $Y \rightarrow Z$ è **non banale** se:

- 1) nessun attributo in Z appartiene a Y ;
- 2) Z non appartiene a Y .

Poiché la DF è un vincolo, una relazione corretta quando soddisfa la DF.

8.3 Forma normale Boyce e Codd

Uno schema $R(x)$ è in forma normale di Boyce e Codd se: è in 1FN; per ogni DF (non banale) $X \rightarrow A$ definita $R(x)$, X contiene una chiave K di $R(x)$, cioè X è superchiave per $R(x)$. Devo vedere le DF e vedere per tutte se a sinistra c'è una chiave.

La *Boyce e Codd* richiede che i concetti in una relazione siano omogenei (solo proprietà direttamente associate alla chiave).

Se una relazione non soddisfa la *Boyce e Codd*, è possibile sostituirla con due o più relazioni normalizzate attraverso il processo di normalizzazione, in cui la relazione viene decomposta in relazioni più piccole, una per ogni concetto indipendente. La decomposizione può essere effettuata producendo tante relazioni quante sono le dipendenze funzionali.

8.3.1 Decomposizione senza perdita

Una relazione r si **decomponе senza perdita** su due relazioni se il join delle proiezioni di r sulle due relazioni è uguale a r stessa (cioè non contiene tuple spurie).

È possibile individuare una condizione che garantisce la decomposizione senza perdita di una relazione:

- sia r una relazione su un insieme di attributi X e siano X_1 e X_2 due sottoinsiemi di X tali che $X = X_1 \cup X_2$;
- inoltre, sia $X_0 = X_1 \cap X_2$;
- Allora, r si **decomponе senza perdita** su X_1 e X_2 se soddisfa la dipendenza funzionale $X_0 \rightarrow X_1$ oppure $X_0 \rightarrow X_2$.

In altre parole, la **decomposizione senza perdita** è garantita se gli attributi comuni nelle relazioni decomposte **contengono una chiave** per almeno una delle relazioni decomposte.

8.3.2 Conservazione delle dipendenze

Una **decomposizione conserva le dipendenze** se ciascuna delle dipendenze funzionali dello schema originario coinvolge attributi che compaiono tutti insieme in uno degli schemi decomposti. In questo modo, è possibile garantire, sullo schema decomposto, il soddisfacimento degli stessi vincoli garantiti dallo schema originario.

8.3.3 Qualità delle decomposizioni

Le decomposizioni dovrebbero sempre soddisfare le proprietà di **decomposizione senza perdita** e **conservazione delle dipendenze**.

- la **decomposizione senza perdita** garantisce la ricostruzione delle informazioni originarie (cioè senza informazioni spurie) a partire da quelle rappresentate nelle relazioni decomposte;
- la **conservazione delle dipendenze** garantisce che le relazioni decomposte hanno la stessa capacità della relazione originaria di rappresentare i vincoli di integrità e quindi di rilevare aggiornamenti illeciti: a ogni aggiornamento lecito sulla relazione originale corrisponde un aggiornamento lecito sulla relazione decomposta.

Dato uno schema che viola una forma normale, l'attività di **normalizzazione** è quindi volta a ottenere una decomposizione che sia senza perdita, conservi le dipendenze e che contenga relazioni in forma normale.

8.4 Terza forma normale

Talvolta non è possibile raggiungere una buona decomposizione in forma normale di Boyce e Codd per questo si ricorre ad una forma normale meno restrittiva che ammette relazioni con alcune anomalie.

Una relazione r è in **terza forma normale** se, per ogni DF (non banale) $X \rightarrow A$ definita su r , è verificata almeno una delle seguenti condizioni:

- ❖ X contiene una chiave K di r ;
- ❖ A appartiene ad almeno una chiave di r .

Inoltre, una relazione è in **terza forma normale** se è già in 2FN (quindi anche in 1FN).

La 3FN è meno restrittiva della forma normale di Boyce e Codd e quindi non offre le stesse garanzie di qualità per una relazione; ha però il vantaggio di essere sempre ottenibile. Si può dimostrare che una qualunque relazione che non soddisfa la 3FN si può decomporre senza perdita e con conservazione delle dipendenze in relazioni in 3FN.

Una relazione che non soddisfa la 3FN si decompone in tante relazioni ottenute per proiezione sugli attributi corrispondenti alle dipendenze funzionali, con l'unica attenzione che alla fine almeno una delle relazioni decomposte contenga una chiave della relazione originaria.

8.5 Prima forma normale

Una tabella è posta in **prima forma normale** se tutti gli attributi che vi compaiono sono semplici, ovvero ogni riga di ciascuna tabella deve poter essere identificata in modo univoco attraverso una **chiave** e non ci devono essere gruppi di attributi che si ripetono (ossia ciascun attributo deve essere definito su un dominio con valori atomici). Quindi vanno evitati attributi multi-valore e attributi composti.

Per togliere gli attributi multi-valore posso creare più tabelle oppure aggiungere una riga per ogni valore dell'attributo.

8.6 Seconda forma normale

Uno schema è in **seconda forma normale** se:

- si trovi già in 1NF;
- tutti i campi non chiave dipendano dall'intera *chiave primaria* (e non solo da una parte di essa).

Per portare una tabella in **seconda forma normale** bisogna:

- individuare per ogni attributo Y che dipende parzialmente dalla chiave il sottoinsieme degli attributi X della chiave da cui dipende;
- costruire una nuova tabella avente X come chiave primaria e Y come attributo;
- togliere Y dalla tabella originaria.

Esempio: nella seguente relazione

Codice	Titolo	Voto	Matricola
INF1	Informatica	7	0988
SIS1	Sistemi	7	0988
INF1	Informatica	8	0325
ITA1	Italiano	8	0546
SIS1	Sistemi	6	0325

l'attributo **Titolo** dipende solo da **codice** e non da **Matricola**
mentre **Voto** dipende da entrambi.

La normalizzazione 2NF darà origine alle seguenti due tabelle.

Codice	Voto	Matricola	Codice	Titolo
INF1	7	0988	INF1	Informatica
SIS1	7	0988	SIS1	Sistemi
INF1	8	0325	INF1	Informatica
ITA1	8	0546	ITA1	Italiano
SIS1	6	0325	SIS1	Sistemi