Basi di Dati

Federico Matteoni

A.A. 2019/20

Indice

Introduzione

Obiettivi del corso Modelli dei dati, linguaggi e sistemi per lo sviluppo di applicazioni che prevedono l'uso di grandi quantità di dati permanenti organizzati in **basi di dati**.

Testo di Riferimento *Fondamenti di Basi di Dati*, A. Albano, G. Ghelli e R. Orsini, Zanichelli. Scaricabile liberamente da `fondamentidibasididati.it`

Terminologia

Base di dati: tecnologia di base, gestione delle attività quotidiane dell'organizzazione e **tema di questo corso**

Data Warehouse, Data Lake, Big Data, Data Science: termini che hanno a che vedere con l'**analisi dei dati** e che non rientrano nei temi trattati nel corso.

Capitolo 1

Costruzione di una base di dati

Cos'è una base di dati? Una **base di dati** è un **insieme organizzato di dati** usati per il supporto allo svolgimento di un'attività (di un ente, azienda, ufficio, persona...)

Qualche esempio

Materie

Titolo	Codice	Syllabus
Basi di Dati	AA024	Progettazione e interrogazione...
Reti di Calcolatori	AA019	Realizzazione e uso di reti, protocollo TCP...

Corsi

Materia	AA	Semestre	Titolare
AA024	2007	1	Albano
AA024	2007	1	Ghelli
AA019	2007	1	Brogi

1.1 Elementi

1.1.1 Figure Coinvolte

Committente

Dirigente

Operatore

Fornitore

Direttore del progetto

Analista

Progettista del DB

Programmatore di applicazioni che usano il DB

Manutenzione e messa a punto del DB – **Gestione del DBMS**

Amministratore del DBMS

1.1.2 Sistemi Informativi

Definizione Un **sistema informativo** di un'organizzazione è una **combinazione di risorse, umane e materiali, e di procedure** organizzate per raccolta, archiviazione, elaborazione e scambio **delle informazioni** necessarie alle attività:

Operative (informazioni di servizio)

Programmazione e **controllo** (informazioni di gestione)

Pianificazione strategica (informazioni di governo)

Esempi di sistemi informativi Un comune

Gestione servizi demografici (anagrafe, stato civile, servizio elettorale e vaccinale) e della rete viaria

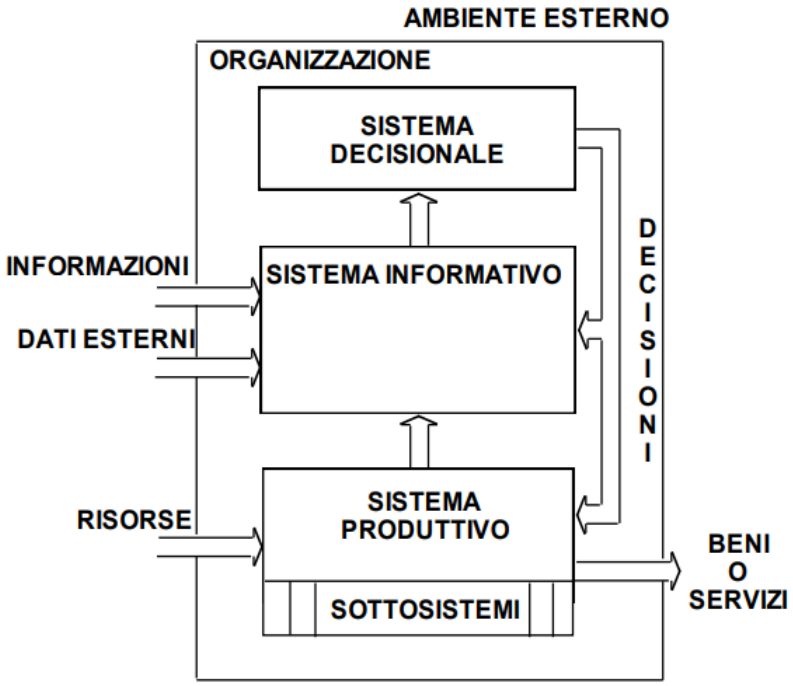
Gestione attività finanziaria secondo la normativa vigente

Gestione del personale per il calcolo della retribuzione in base al tipo di normativa contrattuale

Gestione dei servizi amministrativi e sanitari delle USL

Gestione della cartografia generale e tematica del territorio

Sistema informativo nelle organizzazioni



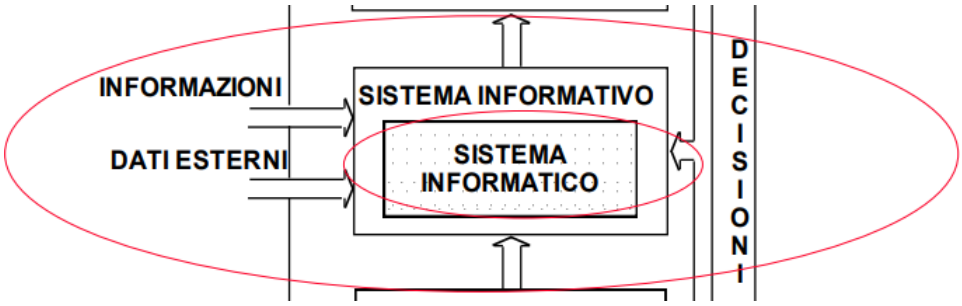
1.1.3 Sistemi Informatici

Sistema Informativo Automatizzato Quella parte del sistema informativo in cui le informazioni sono raccolte, elaborate, archiviate e scambiate usando un **sistema informatico**.

Sistema Informatico Insieme delle tecnologie informatiche e della comunicazione (**ICT**, Information and Communication Technologies) a supporto delle attività di un'organizzazione.

Terminologia

Sistema informativo ⟶ Sistema informativo automatizzato
Sistema informativo automatizzato ⟶ Sistema informatico



1.1.4 Classificazione dei sistemi informatici

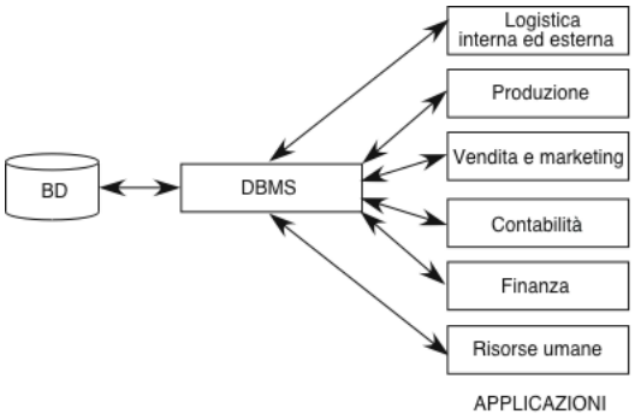
Sistemi Informatici Operativi $\longrightarrow$ Sistemi Informatici Direzionali

Sistemi Informatici Operativi I dati sono organizzati in DB. Le applicazioni si usano per svolgere le classiche attività strutturate e ripetitive dell'azione nelle aree amministrativa e finanziaria: vendite, risorse umane, produzione...

Alcune sigle:

DP Data Processing
EDP Electronic Data Processing

TPS Transaction Processing Systems



DBMS Le caratteristiche del DB sono **garantite da un sistema per la gestione della base di dati** (**DBMS**, Data Base Management System) che ha il controllo dei dati e li rende accessibili agli utenti autorizzati.

OLTP **On-Line Transaction Processing**, modo d'uso principale dei DBMS. Tradizionale elaborazione di transazioni, che realizzano processi operativi per il funzionamento di organizzazioni:

Operazioni predefinite e relativamente semplici

Ogni operazione coinvolge *pochi* dati

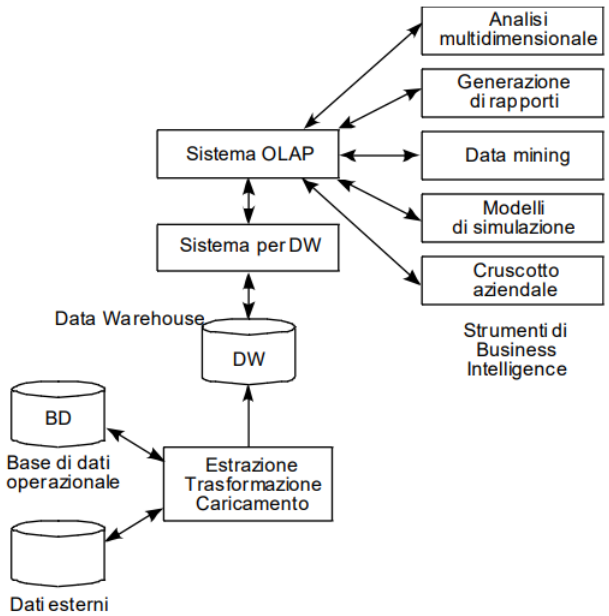
Dati di dettaglio, aggiornati

Sistemi Informatici Direzionali I dati sono organizzati in data warehouse (DW) e gestiti da un opportuno sistema. Le applicazioni, dette di **business intelligence**, sono strumenti di supporto ai processi di controllo delle prestazioni aziendali e di decisione manageriale. Terminologia:

MIS Management Information Systems

DSS Decision Support Systems, data-based o model-based

EIS Executive Information System



OLAP **On-Line Analytical Processing** modo d'uso principale dei DW. Analisi dei dati di supporto alle decisioni:

Operazioni complesse e casuali

Ogni operazione può coinvolgere *molti* dati

Dati aggregati, storici, anche non attualissimi

Differenze tra OLTP e OLAP

	OLTP	OLAP
Scopi	Supporto operatività	Supporto decisioni
Utenti	Molti, esecutivi	Pochi, dirigenti e analisti
Dati	Analitici, relazionali	Sintetici, multidimensionali
Usi	Noti a priori	Poco prevedibili
Quantità di dati per attività	Bassa (decine)	Alta (milioni)
Orientamento	Applicazione	Soggetto
Aggiornamenti	Frequenti	Rari
Visione dei dati	Corrente	Storica
Ottimizzati per	Transazioni	Analisi

1.1.5 Requisiti per l'Analisi dei Dati

Aggregati Non interessa **un** dato, ma la **somma**, la **media**, il **minimo/massimo** di una misura...

Multidimensionale Interessa **incrociare le informazioni**, per analizzarle da punti di vista diversi e valutare i risultati del business per intervenire sui problemi critici o per cogliere nuove opportunità

Diversi livelli di dettaglio Per esempio, una volta scoperto un calo delle vendite in un determinato periodo in una specifica regione, si passa ad un'analisi dettagliata nell'area di interesse per cercare di scoprirne le cause (dimensioni con **gerarchie**)

1.1.6 Big Data

Ampio Big data è un termine ampio riferito a situazioni in cui l'approccio "schema-first" tipico di DB e DW risulta troppo restrittivo o troppo lento.

3 V Volume, Varietà, Velocità

I Big Data sono in genere associati a sistemi NoSQL, machine learning e approcci Data Lake.

1.2 DBMS

Un **DBMS** è un sistema (**software**) in grado di **gestire collezioni di dati** che siano, tra le altre cose:

Grandi

Persistenti, con un periodo di vita indipendente dalle singole esecuzioni dei programmi che le utilizzano

Condivise, usate da applicazioni diverse

garantendo **affidabilità** (resistenza a malfunzionamenti hardware e software-recovery) e **privacy** (con una disciplina e un controllo degli accessi).
Come ogni altro software, un DBMS deve essere **efficiente** (usare al meglio le risorse di spazio e tempo del sistema) ed **efficace** (rendere produttive le attività degli utilizzatori).
Un DBMS offre opportuni linguaggi per:

Definire lo schema di un DB, che va definito prima di creare dati

Scegliere le strutture dati per la memorizzazione

Memorizzare i dati **rispettando i vincoli** definiti nello schema

Recuperare e modificare i dati, interattivamente (**query language**, linguaggio di interrogazione) o da programmi

1.2.1 Dati

I dati permanenti contenuti in un DB sono divisi in due categorie:

Metadati
Descrivono dati sullo schema dei dati, utenti autorizzati, applicazioni, parametri quantitativi...
I metadati sono descritti da uno schema usando il modello dei dati usato dal DBMS e sono interrogabili con le stesse modalità previste dai dati

Dati
Rappresentazioni di certi fatti conformi alle definizioni dello schema. Hanno le seguenti caratteristiche:

Organizzati in **insiemi strutturati e omogenei**, fra i quali sono definite delle **relazioni**. La struttura dei dati e le relazioni sono **descritte nello schema** usando i meccanismi di astrazione del modello dei dati del DBMS.

Sono **molti**, sia in assoluto che rispetto ai metadati, e non possono essere gestiti in memoria temporanea

Sono **accessibili mediante transazioni**, **unità di lavoro atomiche** che **non possono avere effetti parziali**

Sono **protetti** sia **da accesso da parte di utenti non autorizzati**, sia **da corruzione dovuta a malfunzionamenti** hardware o software

Sono **utilizzabili contemporaneamente da utenti diversi**

Il **modello relazionale dei dati** è il più diffuso fra i DBMS commerciali. Il **meccanismo di astrazione** fondamentale è la **relazione** (**tabella**), sostanzialmente un insieme di record dai campi elementari.
Lo schema di una relazione ne definisce il nome e ne descrive la struttura dei possibili elementi della relazione (insieme di attributi con il loro tipo)

Esempio

Definizione del DB

```
create database EsempioEsame
```

Definizione schema

```
create table Esami(Materia char(5), Candidato char(8), Voto int, Lode char(1),
Data char(6))
```

Inserzione dati

```
insert into Esami values ('BDSI1', '080709', 30 'S', '070900')
```

Interrogazione

```
select Candidato from Esami where Materia = "BDSI1" and Voto = 30
       > Candidato
       > 080709
```

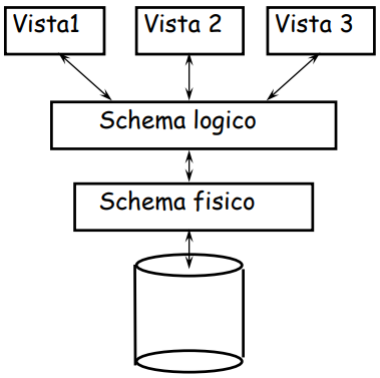
1.2.2 DDL

Data Definition Language Linguaggio per la definizione della base di dati.
Utile distinguere tre diversi livelli di descrizione dei dati (**schemi**):

Livello di **vista logica**

Livello **logico**

Livello **fisico**



Livello Logico Descrive la struttura degli insiemi di dati e delle relazioni fra loro, secondo un certo modello dei dati, senza nessun riferimento alla loro organizzazione fisica nella memoria permanente.
Esempi:

```
Studenti(Matricola char(8), Nome char(20), Login char(8), Anno int, Reddito float)
Corsi(IdeC char(8), Titolo char(20), Credito int)
Esami(Matricola char(8), IdeC char(8), Voto int)
```

Livello Fisico Descrive come vanno organizzati fisicamente i dati nelle memorie permanenti e quali strutture dati ausiliarie prevedere per facilitarne l'uso (schema fisico o interno).
Esempi: relazioni Studenti e Esami organizzate in modo seriale, Corsi organizzata sequenziale con indice, indice su Matricola.

Vista Logica Descrive come deve **apparire la struttura** del DB ad una certa applicazione (**schema esterno** o **vista**). Esempio:

```
InfCorsi (IdeC char(8), Titolo char(20), NumEsami int)
```

Nell'organizzazione di una banca, lo **schema logico** conterrà tutte le tabelle e i dati relativi ai conti correnti, ma anche al personale. Lo schema logico conserva **tutte le informazioni** della banca. Nello **schema esterno** ogni correntista potrà **accedere solo ad alcune informazioni** di suo interesse: quelle del proprio conto corrente.

Indipendenza L'approccio con tre livelli è stato proposto per garantire le proprietà di indipendenza logica e fisica dei dati, fra gli obiettivi più importanti dei DBMS.

Indipendenza fisica: i programmi applicativi non devono essere modificati in seguito a modifiche dell'organizzazione fisica dei dati

Indipendenza logica: i programmi applicativi non devono essere modificati in seguito a modifiche dello schema logico

1.2.3 DML

Data Manipulation Language Linguaggio per l'uso dei dati.
Un DBMS deve prevedere più modalità d'uso per soddisfare esigenze di diverse categorie d'utenti: GUI per accedere ai dati, linguaggio di interrogazione per i non programmatori, linguaggio di programmazione per chi sviluppa le applicazioni, linguaggio di sviluppo per le interfacce delle applicazioni.
Linguaggi vari e interfacce diverse:

Linguaggi testuali interattivi, SQL

Comandi (come quelli del linguaggi interattivo) immersi in un linguaggio ospite, come il C

Comandi (come quelli del linguaggi interattivo) immersi in un linguaggio ad hoc (come PL/SQL) con anche altre funzionalità (come grafici e stampe strutturate)

Interfacce amichevoli

1.2.4 Schemi e Istanze

Schema Descrive la **struttura dei dati**, sostanzialmente invariante nel tempo: le "classi", intestazione delle tabelle

Istanza **Valori attuali** dei dati che possono cambiare anche molto rapidamente: gli "oggetti", il corpo di ciascuna tabella

1.2.5 Meccanismi per il controllo dei dati

Caratteristica molto importante dei DBMS è il tipo di meccanismi usati per garantire le seguenti proprietà

Integrità: mantenimento delle proprietà specificate nello schema

Sicurezza: protezione da usi non autorizzati

Affidabilità: protezione da malfunzionamenti e interferenze dovute all'accesso concorrente di più utenti

1.2.6 Transazioni

Definizione Una **transazione** è una **sequenza di azioni di lettura/scrittura in memoria permanente e di elaborazione dati in memoria temporanea**, con le seguenti proprietà:

Atomicità: le transazioni che terminano prematuramente (**aborted transactions**) sono **trattate dal sistema come se non fossero mai iniziate**. Eventuali effetti sul DB sono **annullati**.

Serializzabilità: esecuzioni concorrenti di più transazioni danno come effetto quello di una esecuzione seriale

Persistenza: le **modifiche** sul DB di una transazione terminata normalmente sono **permanenti**, cioè **non alterabili da malfunzionamenti**

1.3 Progettazione

Progettare Progettare un DB significa **progettare la struttura dei dati e delle applicazioni**. La progettazione dei dati è l'attività più importante e per progettare al meglio i dati è necessario che essi siano un **modello fedele del dominio** in esame. Per questo ora parleremo della **modellazione**.

1.3.1 Modellazione

Definizione Un **modello astratto** è la **rappresentazione formale di idee e conoscenze relative ad un fenomeno**.
Aspetti di un modello:

Il **modello** è la **rappresentazione di certi fatti**.

La **rappresentazione** è **data con un linguaggio formale**.

Il **modello** è **il risultato di un processo di interpretazione**, guidato dalle idee e conoscenze possedute dal soggetto che interpreta.

La stessa realtà può utilmente essere modellata in modi diversi e a diversi livelli di astrazione.

Metodologia di progetto Per garantire prodotti di buona qualità è opportuno seguire una metodologia di progetto, con:

Articolazione delle attività in fasi (**decomposizione**)

Criteri di scelta (**strategie**)

Modelli da rappresentare

Generalità rispetto al problema in esame e agli strumenti a disposizione

Qualità del prodotto

Facilità d'uso

Progettazione della base di dati Suddivisa nelle seguenti fasi:

1. **Analisi** dei requisiti
2. Progettazione **concettuale**
3. Progettazione **logica**
4. Progettazione **fisica**

Ciascuna fase è incentrata sulla modellazione, che discuteremo quindi con riferimento alla problematica della progettazione del DB.

Modello dei dati Insieme di costrutti utilizzati per organizzare i dati di interesse e descriverne la dinamica. Il componente fondamentale è l'insieme dei **meccanismi di strutturazione** (o **costruttori di tipo**). Come nei linguaggi di programmazione, esistono meccanismi che permettono di definire nuovi tipi, così **ogni modello dei dati prevede alcuni costruttori**: per esempio, il **modello relazionale prevede il costruttore *relazione***, che permette di definire insiemi di record omogenei.

1.3.2 Aspetti del problema

Aspetto ontologico

Quale conoscenza del dominio del discorso si rappresenta? Ontologico cioè studio di ciò che si suppone esista nell'universo del discorso e che sia quindi necessario modellare. Cosa si modella:

Conoscenza concreta: i fatti

Conoscenza astratta: la struttura e i vincoli sulla conoscenza concreta

Conoscenza procedurale, comunicazioni: le operazioni base, le operazioni degli utenti, come si comunicherà con il sistema informatico

Ci concentreremo sulla conoscenza concreta e astratta.

Aspetto logico

Con quali meccanismi di astrazione si modella? Il modello dei dati a oggetti.

Modello dei dati Insieme dei meccanismi di astrazione per descrivere la struttura della conoscenza concreta.
Schema: descrizione della **struttura della conoscenza concreta** e dei **vincoli di integrità** usando un particolare modello dei dati.

Useremo come notazione grafica una **variante** dei diagrammi a oggetti (o diagrammi Entità-Relazione, diagrammi ER). Nozioni fondamentali: oggetto, tipo di oggetto, classe, ereditarietà, gerarchia fra tipi e gerarchia fra classi.

Aspetto linguistico

Con quale linguaggio formale si definisce un modello?

Aspetto pragmatico

Come si procede per costruire un modello? Metodologia da seguire nel processo di modellazione, cioè l'insieme di regole finalizzate alla costruzione del modello informatico.

1.3.3 Conoscenza concreta

La conoscenza concreta riguarda i fatti specifici che si vogliono rappresentare:

Entità, sono **ciò di cui interessa rappresentare alcuni fatti o proprietà**. Ad esempio: una descrizione bibliografica di un libro, un libro, un documento, un prestito, un utente della biblioteca...
Le **proprietà** sono **fatti che interessano solo in quanto descrivono caratteristiche di determinate entità**. Ad esempio un indirizzo interessa perché è l'indirizzo di un utente. Hanno delle classificazioni:

Primitiva/strutturata
Obbligatoria/opzionale
Univoca/multivalore
Costante/variabile
Calcolata/non calcolata

Una proprietà è una coppia attributo-valore di un certo tipo. Ogni entità appartiene ad un **tipo** che ne specifica la natura. Ogni proprietà ha associato un **dominio**, l'insieme dei possibili valori.
Una proprietà è **atomica** se il suo valore non è scomponibile, altrimenti è **strutturata**. Inoltre è **univoca** se ha valore unico, altrimenti è **multivalore**, e **totale** (obbligatoria) se ogni entità dell'universo in esame ha per essa un valore specificato, altrimenti è detta **parziale** (opzionale)

Certi fatti possono essere interpretati come proprietà in certi contesti e come entità in altri. Ad esempio, una `DescrizioniBibliografiche` con attributi `autori`, `titolo`, `editore`..., **oppure** un `Autori` con attributi `nome`, `nazionalità`...e `Editori` con `nome`, `indirizzo`...

Collezioni variabili nel tempo di entità omogenee. Ad esempio, la collezione di tutti gli utenti della biblioteca.

Associazioni fra entità

1.3.4 Modellazione ad oggetti

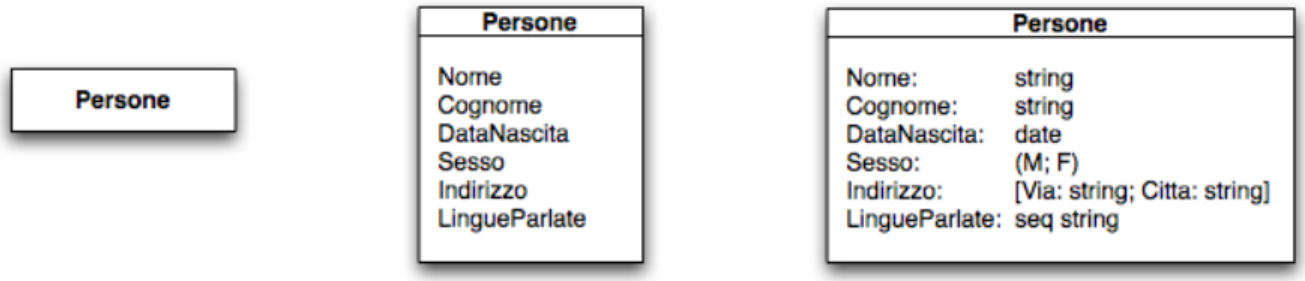
Oggetti Ad ogni entità del dominio corrisponde un oggetto del modello. Un **oggetto** è un'**entità software con stato, comportamento ed identità** che modella un'entità dell'universo.

Stato modellato da un insieme di costanti o variabili con valori di qualsiasi complessità

Comportamento dato da un insieme di procedure locali chiamate **metodi**, che modellano le operazioni di base che riguardano l'oggetto e le proprietà derivabili da altre.

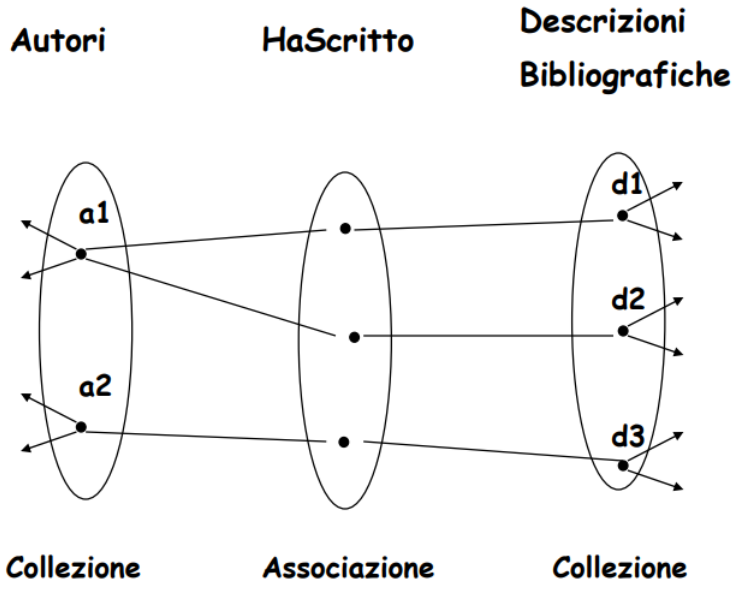
Un oggetto può rispondere a dai **messaggi**, restituendo valori memorizzati nello stato o calcolati con una procedura locale.

Classe **Insieme di oggetti dello stesso tipo**, modificabile con operatori per includere o estrarre elementi dall'insieme. Può essere specificata a diversi livelli.



Tipo oggetto Il primo passo nella costruzione di un modello consiste nella classificazione delle entità del dominio con la definizione dei tipi degli oggetti che la rappresentano.
Un **tipo oggetto definisce l'insieme dei messaggi** (**interfaccia**) **a cui può rispondere un insieme di possibili oggetti**. I nomi dei messaggi sono detti anche attributi degli oggetti.
Nei diagrammi ER i tipi oggetti non si rappresentano, perché l'attenzione è sulle collezioni e sulle associazioni. Tuttavia, la rappresentazione grafica di una collezione indica anche gli attributi del tipo oggetto associato.

Associazioni Un'**istanza di associazione** è un **fatto che correla due o più entità**, stabilendo un legame logico fra loro. Ad esempio, l'utente Tizio ha in prestito una copia della Divina Commedia.
Un'associazione `R(X, Y)` fra due collezioni di entità `X` e `Y` è un **insieme di istanze di associazione** tra elementi di `X` e di `Y` che **varia in generale nel tempo**.
Il prodotto cartesiano $X \times Y$ è il dominio dell'associazione.
Un esempio:



Un associazione è **caratterizzata da due proprietà strutturali**: **molteplicità** e **totalità**.

Vincolo di univocità Un'associazione `R(X, Y)` è **univoca rispetto a** `X` se $\forall$`x` $\in$ `X` $\exists$ al più un elemento di `Y` associato ad `x`.
Se non vale questo vincolo, l'associazione è **multivalore rispetto ad** `X`.

Cardinalità dell'associazione:

`R(X, Y)` è `1:N` se è multivalore su `X` ed univoca su `Y`

`R(X, Y)` è `N:1` se è univoca su `X` e multivalore su `Y`

`R(X, Y)` è `N:M` se è multivalore su `X` e multivalore su `Y`

`R(X, Y)` è `1:1` se è univoca su `X` ed univoca su `Y`

Qualche esempio:

`Frequenta(Studenti, Corsi)` ha cardinalità `N:M`

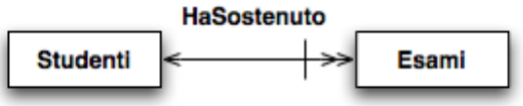
`Insegna(Professori, Corsi)` ha cardinalità `1:N`

`SuperatoDa(Esami, Studenti)` ha cardinalità `N:1`

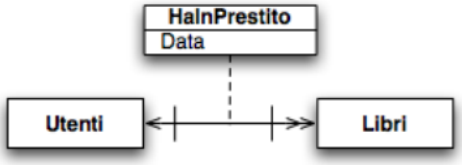
`Dirige(Professori, Dipartimenti)` ha cardinalità `1:1`

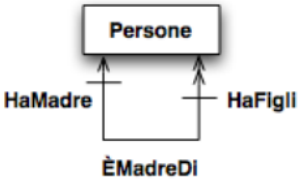
Vincolo di totalità Un associazione `R(X, Y)` è **totale** (o surgettiva) su `X` se $\forall$ `x` $\in$ `X` $\exists$ almeno un elemento di `Y` associato ad `x`.
Se non vale questo vincolo, l'associazione è **parziale rispetto a** `X`.
Ad esempio, `Insegna(Professori, Corsi)` è totale su `Corsi` perché non può esistere un corso senza il corrispondente docente.

Rappresentazione Un'associazione si rappresenta con una linea che collega le classi che rappresentano le due collezioni. La linea è etichettata con il nome dell'associazione, di solito scelto utilizzando un predicato.
L'univocità dell'associazione rispetto ad una classe si rappresenta disegnando una freccia singola sulla linea che esce dalla classe ed entra nella destinazione. L'assenza di tale vincolo è indicata da una freccia doppia.
Similmente, la parzialità è rappresentata da un taglio vicino alla freccia, mentre la totalità è rappresentata dall'assenza del taglio.



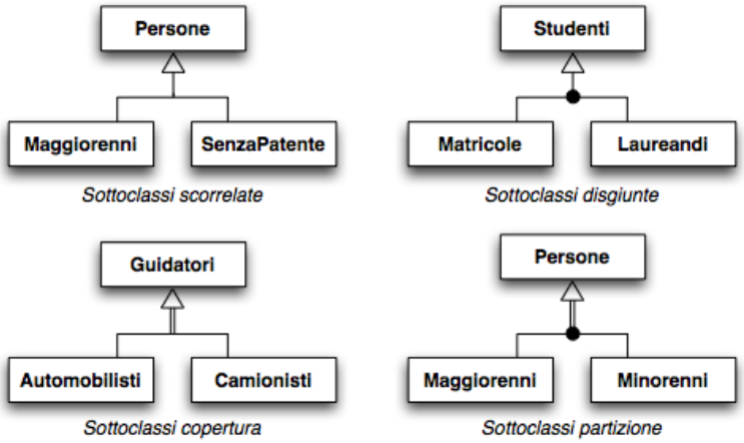
Ogni esame riguarda uno ed un solo studente
Parzialità e assenza di univocità sugli esami
superati da uno studente.





Possono avere proprietà ed essere ricorsive.

1.3.5 Sottoclassi



Vincoli

Vincolo intensionale: C sottoclasse di C' ⇒ tipo degli elementi di C è sottotipo del tipo degli elementi di C'

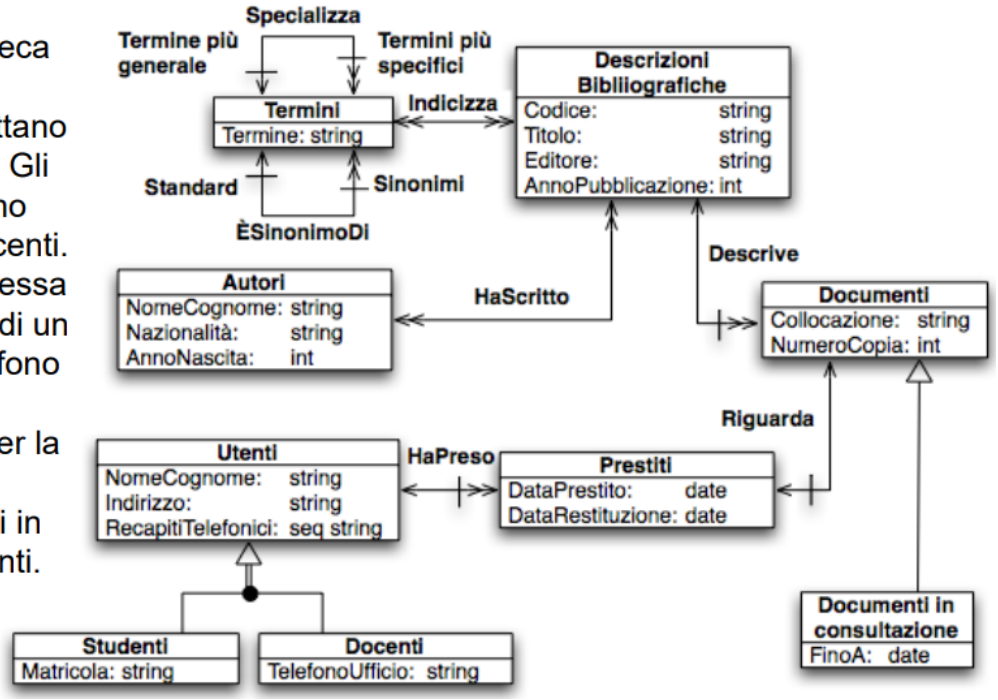
Vincolo estensionale: C sottoclasse di C' ⇒ gli elementi di C sono un sottoinsieme degli elementi di C'

Disgiunzione: ogni coppia di sottoclassi è disgiunta, priva di elementi comuni (pallino nero) **sottoclassi disgiunte**)

Copertura: l'unione degli elementi delle sottoclassi coincide con l'insieme degli elementi della superclasse (freccia con doppia asta) (**sottoclassi copertura**)

1.3.6 Un esempio elaborato

Gli utenti della biblioteca vengono sospesi dal servizio se non rispettano le regole del prestito. Gli utenti regolari possono essere studenti o docenti. Di uno studente interessa anche la matricola e di un docente anche il telefono dell'ufficio.
Alcune opere sono per la sola consultazione e possono essere presi in prestito solo da docenti.



1.3.7 Conoscenza astratta

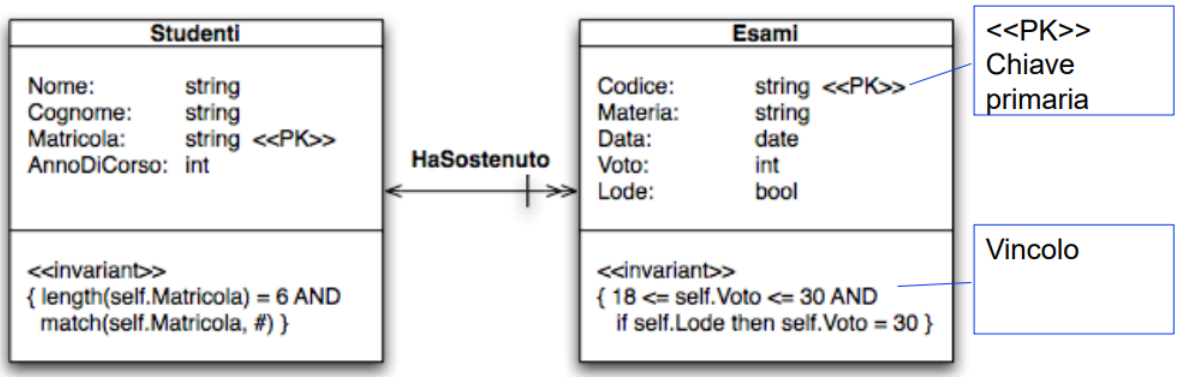
La conoscenza astratta riguarda i **fatti generali che descrivono**

la **struttura della conoscenza concreta**, come collezioni, tipi entità, associazioni...

le **restrizioni sui valori** possibili della conoscenza concreta e sui modi in cui essi possono evolvere nel tempo (**vincoli d'integrità**, statici e dinamici)

le **regole per derivare fatti nuovi** da altri noti

Vincoli Possono essere descritti in **modo dichiarativo** (da preferire), con formule di calcolo dei predicati, oppure mediante controlli da eseguire nelle operazioni.



1.4 Costruzione

1. Analisi dei requisiti → specifica dei requisiti, schemi di settore
2. Progettazione
   - Progettazione **concettuale** (→ schema concettuale), **logica** (→ schema logico), **fisica** (→ schema fisico) dei dati
   - Progettazione delle applicazioni
3. Realizzazione

Spesso consideriamo l'analisi dei requisiti come parte della progettazione.

1.4.1 Analisi dei requisiti

Analizza il sistema esistente e **raccoglie requisiti informali**. Dopodiché **elimina le ambiguità** e la disuniformità, **raggruppando frasi relative a diverse categorie** di dati, vincoli e operazioni.
Costruisce un **glossario**, **disegna lo schema** di settore, **specifica le operazioni** e **verifica la coerenza tra operazioni e dati**.

Documentazione descrittiva In generale, il linguaggio naturale è pieno di ambiguità e fraintendimenti, che bisogna evitare per quanto possibile. Come prima approssimazione si può seguire queste regole:

Studiare e comprendere il sistema informativo ed i bisogni informativi di tutti i settori dell'organizzazione

Scegliere il corretto **livello di astrazione**

Standardizzare la scrittura delle frasi

Suddividere le frasi articolate

Separare le frasi sui **dati** da quelle sulle **funzioni**

Organizzare i concetti e i termini Regole generali

Eliminare le ambiguità, le imprecisioni e la disuniformità: individuare omonimi e sinonimi e unificare i termini

Riorganizzare le frasi per **concetti**, ovvero ottenendo diverse categorie di dati, vincoli e operazioni

Costruire un **glossario** dei termini

Disegnare lo schema

Specificare le operazioni

Verificare la coerenza fra le operazioni e i dati

1.5 Modello Relazionale

Origini Proposto da E. F. Codd nel 1970 per favorire l'indipendenza dei dati, disponibile in DBMS reali dal 1981 (non è facile implementare l'indipendenza con efficienza e affidabilità). Si basa sul **concetto matematico di relazione** con una variante, naturalmente rappresentata come **tabella**.

1.5.1 Relazione matematica

Dalla teoria degli insiemi Dati n insiemi anche non distinti $D_1, \ldots, D_n$.
Il **prodotto cartesiano** $D_1 \times \ldots \times D_n$ è l'insieme di tutte le n-uple $(d_1, \ldots, d_n)$ tali che $d_1 \in D_1, \ldots, d_n \in D_n$

Una **relazione matematica** su $D_1, \ldots, D_n$ **è un sottoinsieme di** $D_1 \times \ldots \times D_n$, con $D_1, \ldots, D_n$ detti **domini della relazione**.

Un esempio Dati $D_1 = \{a, b\}$ e $D_2 = \{x, y, z\}$.
Il prodotto cartesiano è l'insieme $D_1 \times D_2 = \{(a, x), (a, y), (a, z), (b, x), (b, y), (b, z)\}$
Una relazione r potrebbe essere $r \subset D_1 \times D_2 = \{(a, x), (a, z), (b, y)\}$

Proprietà Una relazione matematica è un insieme di n-uple ordinate $(d_1, \ldots, d_n)$ tali che $d_1 \in D_1, \ldots, d_n \in D_n$.
Osservazioni: una relazione è un insieme, quindi

non c'è ordinamento fra le n-uple

le n-uple sono distinte

ciascuna n-upla è ordinata, cioè l'i-esimo valore proviene dall'i-esimo dominio

Tabelle 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 fra loro

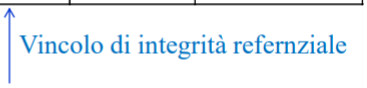
In una tabella che rappresenta una relazione **l'ordinamento** tra le righe e l'ordinamento tra le colonne **è irrilevante**

1.5.2 Valori

Il modello relazionale è basato sui valori Ciò significa che i riferimenti fra dati in relazioni diverse sono rappresentati per mezzo di valori dei domini che compaiono nelle n-uple.

Studenti

Nome	Matricola	Provincia	AnnoNascita
Isaia	071523	PI	1982
Rossi	067459	LU	1984
Bianchi	079856	LI	1983
Bonini	075649	PI	1984



Esami

Materia	Candidato*	Data	Voto
BD	071523	12/01/06	28
BD	067459	15/09/06	30
FP	079856	25/10/06	30
BD	075649	27/06/06	25
LMM	071523	10/10/06	18

Vantaggi

Indipendenza delle strutture fisiche che possono cambiare dinamicamente, che potremmo avere anche con puntatori di alto livello. La rappresentazione logica dei dati (che è costituita dai soli valori) non fa riferimento a quella fisica.

Si **rappresenta solo ciò che è rilevante** dal punto di vista dell'applicazione

I **dati** sono **portabili** più facilmente da un sistema all'altro

I **puntatori** sono **direzionali**

1.5.3 Meccanismi

Definizione I meccanismi per definire una base di dati con il modello relazionale sono l'**ennupla** e la **relazione**.

Tipo ennupla Un tipo ennupla T è un insieme finito di coppie ⟨attributo, tipo elementare⟩
Se T è un tipo ennupla, R(T) è lo schema della relazione R, quindi **lo schema di un DB è l'insieme di schemi di relazione $\mathbf{R}_i(\mathbf{T}_i)$**. Un'**istanza** di uno schema R(T) è un insieme finito di ennuple di tipo T.

Informazione incompleta Per rappresentare un'informazione incompleta (es.: l'assenza del secondo nome) **non bisogna usare elementi del dominio** come lo 0, stringa vuota, "99"....
Questo perché potrebbero non esistere valori "non utilizzati", e se esistono potrebbero diventare significativi. Inoltre, in fase di utilizzo (nei programmi) bisognerebbe tenere conto ogni volta del "significato" di questi valori.

Il **valore nullo** denota l'**assenza di un valore del dominio e non è un valore del dominio**.
Quindi t[A] per ogni attributo A è un valore del dominio dom(A) oppure è il valore nullo NULL.
Si possono (e **devono**) imporre restrizioni sulla presenza di valori sulli.

Vincoli d'Integrità Esistono istanze di DB che, nonostante siano sintatticamente corrette, non rappresentano informazioni possibili per l'applicazione e che quindi **generano informazioni prive di significato**. Ad esempio un voto di 32, o due studenti con la stessa matricola.
Uno **schema relazionale è costituito da un insieme di schemi di relazione e un insieme di vincoli d'integrità** sui possibili valori delle estensioni delle relazioni.
Un **vincolo d'integrità** è una **proprietà che deve essere soddisfatta dalle istanze che rappresentano informazioni corrette** per l'applicazione.

Vincoli Intrarelazionali: sono vincoli che **devono essere rispettati dai valori contenuti nella relazione considerata**. Vincoli **sui valori** (o **di dominio**), vincoli **di ennupla**.

Vincoli Interrelazionali: sono vincoli che **devono essere rispettati da valori contenuti in relazioni diverse**.

Chiave Informalmente, una **chiave** è un **insieme di attributi che identificano le ennuple di una relazione**. Formalmente, **un insieme K di attributi è *superchiave* per** r **se** r ***non*** **contiene due ennuple distinte** t_1 **e** t_2 **con** $t_1[K] = t_2[K]$.
K è chiave per r **se è superchiave minimale per** r, cioè se non contiene altre superchiavi.
Un esempio classe è la matricola: è superchiave ed è un solo attributo, quindi è minimale.

Una relazione non può contenere ennuple distinte ma con valori uguali. Ogni relazione ha **sicuramente** come superchiave l'insieme di tutti gli attributi su cui è definita, quindi ogni relazione ha (almeno) una chiave.
L'esistenza delle chiavi garantisce l'accesso di ciascun dato della base di dati e permettono di correlare i dati in relazioni diverse.
Un valore nullo in una chiave non permette di identificare le ennuple o realizzare i riferimenti con altre relazioni. Una **chiave primaria** è una chiave su cui non sono ammessi valori nulli: si denota sottolineando il nome dell'attributo es. matricola.

Integrità referenziale Le informazioni in relazioni diverse sono correlate attraverso valori comuni. In particolare, **vengono prese in considerazione i valori delle chiavi primarie**, quindi le **correlazioni devono essere coerenti**.
Un **vincolo di integrità referenziale** (foreign key) **fra gli attributi X di una relazione** $\mathbf{R_1}$ **e un'altra relazione** $\mathbf{R_2}$ **impone ai valori su X in** $\mathbf{R_1}$ **di comparire come valori della chiave primaria di** $\mathbf{R_2}$.

1.6 Trasformazione di schemi

Progettazione logica L'obiettivo della progettazione logica è **tradurre lo schema concettuale in uno schema logico relazionale**, che rappresenti gli stessi dati, **in maniera corretta ed efficiente**. Questo richiede una ristrutturazione del modello concettuale.

1.6.1 Progettazione logica relazionale

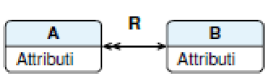
Passaggi La progettazione di uno schema ad oggetti in uno schema relazionale avviene seguendo questi passaggi:

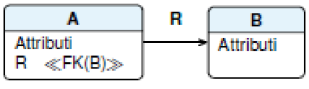
1. rappresentazione delle associazioni 1:1 e 1:N
2. rappresentazione delle associazioni N:M o non binarie
3. rappresentazione delle gerarchie d'inclusione
4. identificazione delle chiavi primarie
5. rappresentazione degli attributi multivalore

Obiettivo Rappresentare le **stesse informazioni**, **minimizzando la ridondanza** e **producendo uno schema comprensibile** che faciliti la scrittura e la manutenzione delle applicazioni.

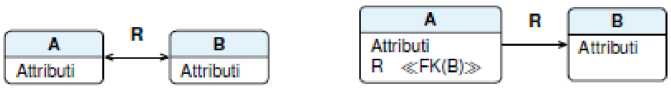
1.6.2 Rappresentazione delle associazioni

Uno a molti Si rappresentano aggiungendo agli attributi della relazione rispetto a cui l'associazione è univoca la chiave esterna che riferisce l'altra relazione.

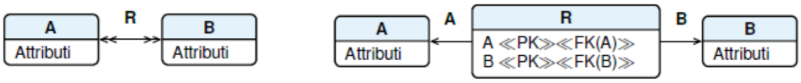




Uno a uno Si rappresentano aggiungendo la chiave esterna ad una qualunque delle due relazioni che riferisce l'altra relazione. Nel caso di un vincolo di totalità, la chiave esterna viene aggiunta alla relazione rispetto cui l'associazione è totale.



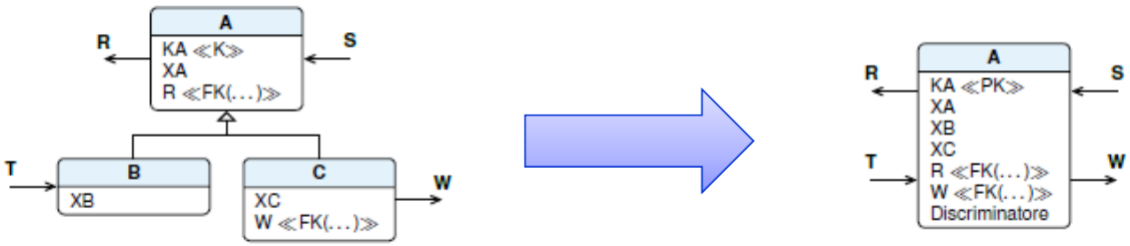
Molti a molti Si rappresenta aggiungendo allo schema una nuova relazione contenente due chiavi esterne che riferiscono le due relazioni coinvolte. La chiave primaria di questa relazione è costituita dall'insieme di tutti i suoi attributi.



1.6.3 Rappresentazione delle gerarchie fra classi

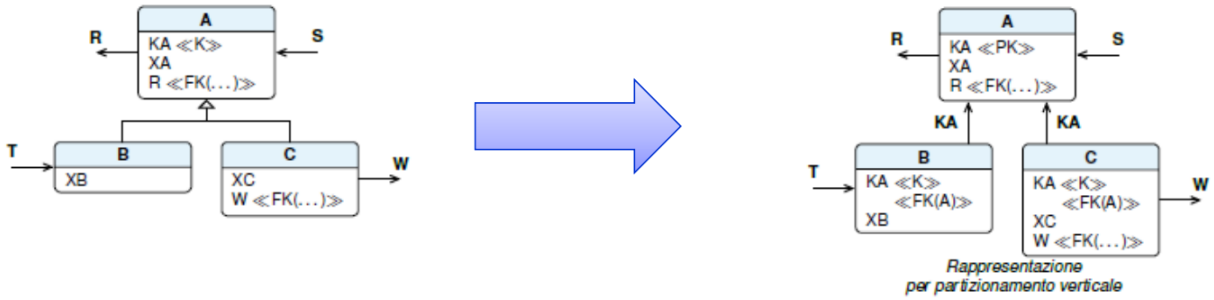
Il modello relazionale non può rappresentare direttamente le generalizzazioni. Bisogna eliminare le gerarchie, sostituendole con classi e relazioni:

Relazione unica: accorpamento delle figlie della generalizzazione nel genitore
Se A_0 è la classe genitore di A_1 e A_2, allora A_1 e A_2 vengono eliminate e accorpate ad A_0. Ad A_0 viene aggiunto un attributo (**discriminatore**) che indica da quale delle classi figlie deriva una certa istanza, e **gli attributi di A_1 e A_2 vengono assorbiti da A_0**, assumendo valore nullo sulle istanze provenienti dall'altra classe.



Partizionamento orizzontale: accorpamento del genitore della generalizzazione nelle figlie
La classe genitore A_0 viene eliminata e le classi figlie A_1 e A_2 ereditano le proprietà (attributi, identificatore e relazioni) della classe genitore. Le relazioni della classe genitore vengono sdoppiate, coinvolgendo ciascuna delle figlie.
Divide gli elementi della superclasse in più relazioni diverse, per cui **non è possibile mantenere un vincolo referenziale verso la superclasse stessa**. Quindi, **non si usa se nello schema relazionale grafico c'è una freccia che entra nella superclasse** (come la $\leftarrow^S$ nell'esempio sopra, che entra nella superclasse A).

Partizionamento verticale: sostituzione della generalizzazione con relazioni.
La generalizzazione si trasforma in due associazioni uno ad uno che legano rispettivamente la classe progenitore con le classi figlie. In questo caso, **non c'è un trasferimento di attributi o di associazioni** e le classi figlie A_1 e A_2 sono identificate esternamente dalla classe genitore A_0.
Nello schema ottenuto si aggiungono dei vincoli: ogni occorrenza di A_0 non può partecipare contemporaneamente alle due associazioni e se la generalizzazione è totale, deve partecipare almeno una delle due.



Rappresentazione per partizionamento verticale

Capitolo 2

Algebra Relazionale

2.1 Linguaggi

Linguaggi per i DB

DDL Data Definition Language, per le **operazioni sullo schema**.
Operazioni di creazione, cancellazione e modifica di schemi di tabelle, creazione viste, creazione indici...

DML Data Manipulation Language, per le **operazioni sui dati**.

Data Query Language, per le **query**, cioè l'**interrogazione del DB**

Aggiornamento dati, per inserimento, cancellazione e modifica dei dati.

Linguaggi relazionali

Algebra relazionale: insieme di operatori su relazioni che danno come risultato altre relazioni.
Non si usa come linguaggio di interrogazione dei DBMS ma come rappresentazione interna delle interrogazioni.

Calcolo relazionale: linguaggio dichiarativo di tipo logico da cui è stato derivato l'SQL.

2.2 Operatori

Unione, intersezione e differenza

Ridenominazione

Selezione

Proiezione

Join (naturale, prodotto cartesiano, theta-join)

Sono **operatori insiemistici**: **le relazioni sono insiemi** e i risultati devono essere relazioni a loro volta. L'unione, intersezione e differenza sono applicabili solamente a relazioni definite sugli stessi attributi, cioè **possono operare solo su tuple uniformi**.

2.2.1 Ridenominazione

Operatore monadico Un solo argomento, modifica lo schema lasciando inalterata l'istanza dell'operando. Si indica con la lettera ρ, esempio: $\rho\ nomecolonna \leftarrow nuovonome$

2.2.2 Proiezione

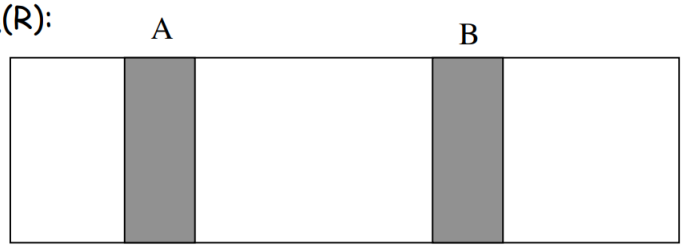
Operatore monadico Produce un risultato che ha **parte degli attributi** dell'operando e **contiene ennuple cui contribuiscono tutte le ennuple dell'operando ristrette agli attributi nella lista**. Esempio $\pi_{lista\ attributi}(operando)$
$\pi_{A_1 \ldots A_n}(R)$
Contiene **al più** tante ennuple quante l'operando, ma può contenerne meno. Se X è superchiave di R, allora π_X(R) contiene esattamente tante ennuple quante R. Se X non è superchiave, **potrebbero esistere valori ripetuti su quegli attributi**, che quindi **vengono rappresentati una sola volta**.

2.2.3 Selezione

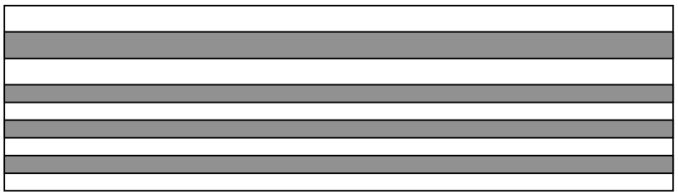
Operatore monadico Produce un risultato con **lo stesso schema dell'operando**, **contiene un sottoinsieme delle ennuple dell'operando** cioè quelle che **soddisfano una condizione** espressa combinando con i connettivi logici $\wedge$, $\vee$, $\neg$, condizioni atomiche del tipo $A\Theta B$ o $A\Theta c$ dove Θ è un operatore di confronto, A e B sono attributi su cui l'operatore Θ abbia senso e c sia una costate compatibile col dominio di A.
Viene denotata con $\sigma_{condizione}(operando)$, ad esempio $\sigma_{Stipendio>50 \wedge Filiale='Milano'}$(Impiegati)
Per riferirsi ai valori nulli si usano apposite condizioni IS NULL e IS NOT NULL.

$$\sigma_{eta>30}(\text{Persone}) \cup \sigma_{eta\leq 30}(\text{Persone}) \cup \sigma_{eta\ IS\ NULL}(\text{Persone})$$
$$=$$
$$\sigma_{eta>30\ \vee\ eta\leq 30\ \vee\ eta\ ISNULL}(\text{Persone})$$
$$=$$
$$\text{Persone}$$

- Proiezione $\pi_{A,B}(R)$:



- Restrizione $\sigma_{Cond}(R)$:



2.2.4 Join

Giunzione Combinando selezione e proiezione possiamo estrarre informazioni da una relazione, ma **non possiamo correlare informazioni presenti in relazioni diverse**.
Il **join** è l'operatore più interessante dell'algebra relazionale perché consente di correlare i dati in relazioni diverse.

Join naturale Operatore binario (generalizzabile) che **correla dati** in relazioni diverse **sulla base di valori uguali in attributi con lo stesso nome**.
Produce un risultato sull'unione degli attributi degli operandi con **ennuple ottenute combinando le ennuple degli operandi con valori uguali sugli attributi in comune**.
Dati $R_1(X_1)$, $R_2(X_2)$, allora $R_1 \bowtie R_2$ è una relazione su $X_1 \cup X_2$
$R_1 \bowtie R_2 = \{t \text{ su } X_1 \cup X_2 \mid \exists t_1 \in R_1 \wedge t_2 \in R_2 \text{ con } t[X_1] = t_1 \text{ e } t[X_2] = t_2\}$

Cardinalità Dati $R_1(A, B)$ e $R_2(B, C)$, il join contiene un numero di ennuple compreso fra 0 e $|R_1| \cdot |R_2|$

$$0 \leq |R_1 \bowtie R_2| \leq |R_1| \cdot |R_2|$$

Se il join è completo, allora contiene un numero di ennuple almeno uguale al massimo tra $|R_1|$ e $|R_2|$. Se il join coinvolge una chiave B di R_2, allora $0 \leq |R_1 \bowtie R_2| \leq |R_1|$.
Se coinvolge una chiave B di R_2 e un vincolo di integrità referenziale tra attributi di R_1 e la chiave di R_2,
allora $|R_1 \bowtie R_2| = |R_1|$

Join esterno Estende con valori nulli le ennuple che verrebbero tagliate fuori da un join interno.
Esiste in tre versioni:

Sinistro $R \overleftarrow{\bowtie} S$, mantiene tutte le ennuple del primo operando estendendole con valori nulli se necessario.

Destro $R \overrightarrow{\bowtie} S$, idem ma del secondo operando.

Completo, idem ma di entrambi gli operandi.

Prodotto cartesiano Un join naturale senza attributi in comune: contiene sempre un numero di ennuple pari al prodotto delle cardinalità degli operandi.

Theta-join Il prodotto cartesiano ha senso solo se seguito da una selezione $\sigma_{condizione}(R_1 \bowtie R_2)$, che viene abbreviato con $R_1 \bowtie_{condizione} R_2$

Self-join Non si potrebbe fare una join del tipo Genitore $\bowtie$ Genitore, perché ritornerebbe la stessa tabella Genitore poiché tutti gli attributi coincidono. Torna utile effettuare una ridenominazione del tipo $\rho_{Nonno,Genitore \leftarrow Genitore,Figlio}$(Genitore) per poi effettuare una natural join del risultato con Genitore.

2.2.5 Raggruppamento

Con l'operatore ${}_{\{A_i\}}\gamma_{\{f_i\}}$(R) si effettura il raggruppamento.

A_i sono gli **attributi** di R

f_i sono le **espressioni** che usano **funzioni di aggregazione** (min, max, count, sum...)

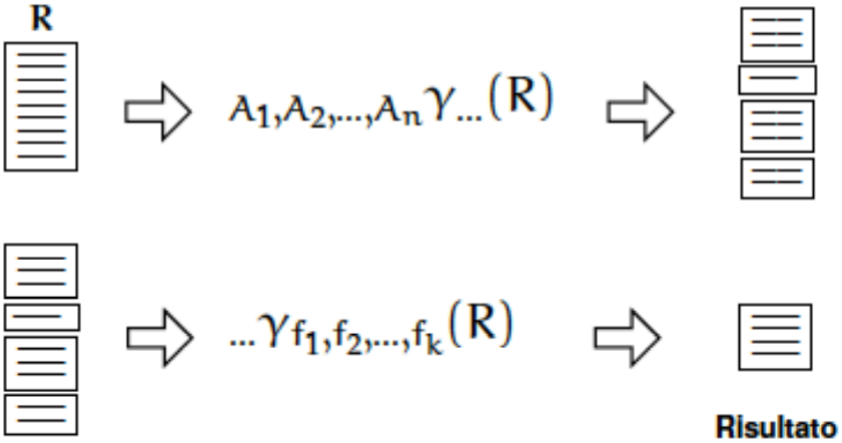
Il valore del raggruppamento è calcolato come segue:

si partizionano le ennuple di R mettendo nello stesso gruppo tutte le ennuple con valori uguali degli A_i (si **raggruppa per** A_i)

si **calcolano le espressioni** f_i per ogni gruppo

per ogni gruppo, si **restituisce una sola ennupla con attributi i valori** degli A_i e delle espressioni f_i

$$ {}_{\{A_i\}}\gamma_{\{f_i\}}(R) $$



Capitolo 3

Interrogazione di una base di dati

SQL L'interrogazione di una base di dati è uno degli aspetti più importanti del linguaggio SQL. I comandi di interrogazione, o **query**, permettono di effettuare una ricerca dei dati presenti nel database che soddisfano particolari condizioni richieste dall'utente.

```
SELECT  s.Nome, e.Data
FROM    Studenti s, Esami e
WHERE   e.Materia = 'BD' AND e.Voto = 30 AND e.Matricola = s.Matricola

SELECT  s.Nome AS Nome, 2020 - s.AnnoNascita AS Eta, 0 AS NumeroEsami
FROM    Studenti s
WHERE   NOT EXISTS(SELECT * FROM Esami e WHERE e.Matricola = s.Matricola)
```

Storia Definito nel 1973, SQL è oggi il linguaggio universale dei sistemi relazionali. Ci sono vari standard (SQL-84, SQL-89... fino a SQL-1999 ANSI/ISO ad oggetti) ed è composto da DDL, DML e **query language**.

Select From Where SQL è un **calcolo su multinsiemi**. Il comando base dell'SQL è:

```
SELECT [DISTINCT] Attributo{, Attributo}
FROM Tabella [Ide]{, Tabella [Ide]}
[WHERE Condizione]
```

La semantica è **prodotto**, **restrizione**, **proiezione**. Un attributo A di una tabella "R x" si denota come A, R.A oppure x.A

```
SELECT ListaAttributi
FROM ListaTabelle
[WHERE Condizione]
```

La query considera il **prodotto cartesiano tra le tabelle in** `ListaTabelle` (**join**).
Fra queste, **seleziona solo le righe che soddisfano la** `Condizione` (**selezione**).
Infine, **valuta le espressioni specificate nella target list** `ListaAttributi` (**proiezione**).
La SELECT quindi implementa gli operatori di proiezione, selezione e join dell'algebra relazionale.

3.0.1 SELECT

Proiezione Specifica la target list, cioè corrisponde a scegliere gli attributi della/e tabella/e interessate. Implementa quindi l'operazione di **proiezione** dell'algebra relazionale.

3.0.2 FROM

Tabelle Ha lo scopo di scegliere quali sono le tabelle del database da cui vogliamo estrarre le nostre informazioni. Nel caso in cui le tabelle elencate siano due, la FROM insieme alla WHERE implementa il theta-join.

3.0.3 WHERE

Selezione Serve a scegliere le righe della tabella che soddisfano una certa condizione. In questo modo la clausola WHERE implementa la **selezione** dell'algebra relazionale.

Condizioni In SQL sono disponibili una serie di condizioni a seconda del tipo di dato da confrontare, oltre ai IS NULL e IS NOT NULL per i dati mancanti.
In particolare, con l'operatore LIKE si possono effettuare ricerche con wildcard: % per zero o più caratteri, _ per un carattere. Per esempio, `WHERE Nome LIKE %a` ricercherà tutti i valori del campo `Nome` che finiscono per `a`, oppure `WHERE Sequenza LIKE %G___G%` seleziona i valori dove compaiono due `G` separate da 3 caratteri.
Si possono inserire anche dei simboli escape, ad esempio se vogliamo cercare valori in cui compare `%` si può scrivere `WHERE Sconto LIKE _5#% ESCAPE #`, così da trovare tutti i valori sconto con 5 nelle unità.

3.1 Ordinamento e aggregazione

Ordinamento Il risultato di una SELECT si può ordinare in base ad un attributo e in maniera crescente o decrescente

SELECT `ListaAttributi`
FROM `ListaTabelle`
WHERE `Condizione`
ORDER BY `Attributo` [**ASC**/**DESC**]

Le righe verranno ordinate in base al campo Attributo in maniera crescente (ASC) o descrescente (DESC). L'ordinamento è quello più naturale sul dominio dell'attributo (numerico, alfabetico...).
Si possono anche fare ordinamenti multipli, ad esempio se si vuole ordinare i dati in base ad una certa chiave (attributo) e poi ordinare i dati che coincidono su quella chiave in base ad un'altra chiave (altro attributo).

...
ORDER BY `Attributo1` [**ASC**/**DESC**] { , `Attributon` [**ASC**/**DESC**] }

Verranno ordinati prima per Attributo1, le righe coincidenti su Attributo1 verranno ordinate per Attributo2....

Aggregazione Nella target list possiamo avere anche espressioni che calcolano valori a partire da insiemi di ennuple e che restituiscono una tabella molto particolare, costituita da un **singolo valore scalare**.
Sono 5 gli **operatori aggregati** previsti da SQL2:

`COUNT()`, conteggio: restituisce il numero di righe della tabella o il numero di valori di un particolare attributo. Con `(*)` conta tutte le righe selezionate, con `ALL` conta tutti i valori non nulli delle righe selezionate, `DISTINCT` conta tutti i valori non nulli distinti delle righe selezionate. Di default è `ALL`.

`MIN()`, minimo. L'argomento può essere una funziona aritmetica.

`MAX()`, massimo. L'argomento può essere una funziona aritmetica.

`SUM()`, somma. Le specifiche `ALL` e `DISTINCT` permettono di sommare tutti i valori non nulli o tutti i valori distinti. Anche qua di default è `ALL`.

`AVG()`, media (dei valori non nulli). `ALL` e `DISTINCT` per calcolare la media fra tutti i valori o fra quelli distinti, default `ALL`.

Non è possibile utilizzare, in una stessa SELECT, una proiezione su alcuni attributi della tabella e operatori aggregati sulla stessa tabella. Si possono, invece, fare operazioni aggregate su colonne diverse della stessa tabella.

Raggruppamento A volta può essere richiesto di calcolare operatori aggregati non per l'intera tabella ma raggruppando le righe i cui valori coincidono su un certo attributo: possiamo scrivere `GROUP BY Attributo`.
Con `HAVING` si possono applicare condizioni sul valore aggregato per ogni gruppo.
Attributo → `WHERE`
Operatore aggregato → `HAVING`

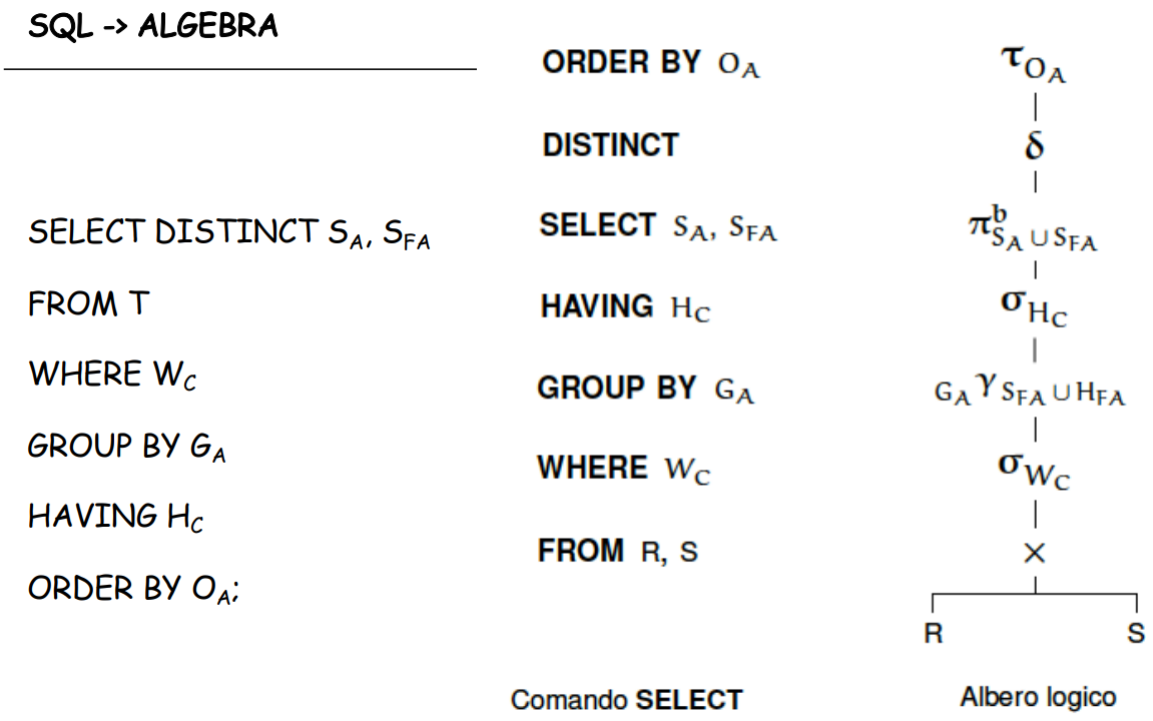
3.2 Semantica

La query è innanzitutto eseguita senza operatori aggregati e senza `GROUP BY`.
Quindi il risultato è diviso in sottoinsiemi aventi gli stessi valori per gli attributi elencati in `GROUP BY`.
Quindi l'operatore aggregato è calcolato su ogni sottoinsieme.

Osservazione Quando si effettua un **raggruppamento**, questo **deve essere effettuato su tutti gli elementi della target list che non sono operatori aggregati** (ossia sull'insieme degli attributi puri).
Questo perché **nel risultato deve apparire una riga per ogni gruppo**.

Esempio di query `HAVING` può essere usato solo in presenza di un `GROUP BY` e dopo di esso.

```
SELECT    Citta, Avg(Voto)
FROM      EsamiBD
WHERE     Eta < 21
GROUP BY  Citta
HAVING    AVG(Voto) > 26
```



3.3 Subquery

Una **subquery** un comando SELECT racchiuso tra parantesi tonde ed inserito all'interno di un altro comando SQL (ad esempio un'altra query).
Possono essere usate nei seguenti casi:

- in espressioni di confronto
- in espressioni di confronto quantificato
- in espressioni IN
- in espressioni EXISTS
- nel calcolo di espressioni

Tipologie Ci sono tre tipologie di subquery:

Subquery Scalare: comando SELECT che restituisce **un solo valore**
Es: `SELECT MAX(Cilindrata) FROM Veicoli`

Subquery di Colonna: comando SELECT che restituisce **una colonna**
Es: `SELECT CodCateogira FROM Veicoli`

Subquery di Tabella: comando SELECT che restituisce **una tabella**
Es: `SELECT Targa, CodMod, Posti FROM Veicoli`

3.4 Quantificazione

Tutte le interrogazioni su di un'associazione multivalore vanno quantificate.

La query "gli studenti che hanno preso 30" è **ambigua**.

Gli studenti che hanno preso **sempre** 30: **universale**

Gli studenti che hanno preso **almeno un** 30: **esistenziale**

Gli studenti che **non** hanno preso **qualche** 30: **universale**

Gli studenti che **non** hanno preso **sempre** 30: **esistenziale**

Universale negata = esistenziale
Esistenziale negata = universale

ANY, ALL, EXISTS Le condizioni in SQL permettono il confronto fra un attributo ed il risultato di una subquery che restituisce una colonna od una tabella.

Operatore **scalare** (ANY | ALL) `SELECT...`
ANY: il predicato è vero se **almeno uno dei valori restituiti dalla subquery soddisfano la condizione**
ALL: il predicato è vero se **tutti i valori restituiti dalla subquery soddisfano la condizione**

Quantificatore **esistenziale** EXISTS `SELECT...`
Il predicato è vero se **la subquery restituisce almeno una tupla**.

3.5 Unione, Intersezione, Differenza

A volte può essere utile poter ottenere un'unica tabella contenente alcuni dei dati contenuti in due tabelle omogenee, ossia con attributi definiti sullo stesso dominio.
La SELECT da sola non permette di fare questo tipo di operazioni su tabelle. Esistono per questo dei costrutti espliciti che utilizzano le parole chiave UNION, INTERSECT, EXCEPT (o MINUS).
Tali operatori lavorano sulle tabelle come se fossero insiemi di righe, dunque i duplicati vengono eliminati anche dalle proiezioni (a meno di non specificare ALL).
Questi operatori vanno in mezzo a due SELECT:

SELECT...
(**UNION** | **INTERSECT** | **EXCEPT**) [**ALL**]
SELECT...

UNION Realizza l'operazione di unione definita nell'algebra relazionale. Utilizza come operandi le due tabelle risultanti da comandi SELECT e restituisce una terza tabella che contiene **tutte le righe della prima e della seconda tabella**.
Nel caso in cui dall'unione e dalla proiezione risultassero delle righe duplicate, UNION ne mantiene una sola copia (a meno di aver specificato ALL dopo UNION).
Mantiene i nomi delle colonne del primo operando. Quindi, se si vuole ridenominare, è bene ridenominare tutte le colonne che vogliamo.

INTERSECT Utilizza come operandi due tabelle risultati dai comandi SELECT e restituisce una tabella che contiene **le righe comuni alle due tabelle iniziali**. Anche qua con ALL mantiene i duplicati, e realizza l'interrsezione dell'algebra relazionale.

EXCEPT Utilizza come operandi due tabelle ottenute mediante due SELECT, ed ha come risultato una nuova tabella che contiene **tutte le righe della prima che non si trovano nella seconda**. Realizza la differenza dell'algebra relazionale, ed anche qua si possono mantenere i duplicati utilizzando ALL.

Capitolo 4

Modifica di una base di dati

4.1 Modifica dei dati

Data Manipulation Language Introduciamo ora il DML, ossia il linguaggio SQL che serve per **inserire, modificare e cancellare i dati** del database ma anche per **interrogare il database** per estrarne i dati.
Inizieremo descrivendo le istruzioni che servono a inserire, cancellare e modificare i dati, per poi introdurre le istruzioni per estrarre dal database le informazioni che ci interessano.

INSERT Per inserire un nuovo dato in una tabella si usa INSERT INTO... VALUES

```
INSERT INTO Tabella [(ListaAttributi)] (VALUES (ListaValori) | Subquery)
```

Update Si possono aggiornare alcuni dati con UPDATE

```
UPDATE Tabella SET Attributo = Expr{, Attributo = Expr} WHERE Condizione
```

DELETE Per cancellare righe dalle tabelle si usa la DELETE

```
DELETE FROM Tabella [WHERE Condizione]
```

Per eliminare un elemento bisogna individuare quale, stabilito con la clausola WHERE con la condizione che individua l'elemento (o gli elementi) da eliminare.
Un particolare elemento può essere individuato dal suo valore nella chiave primaria.

4.2 Definizione degli oggetti

Data Definition Language Introduciamo il DDL, cioè il linguaggio SQL che consiste nell'**insieme delle istruzioni che permettono la creazione, modifica e cancellazione delle tabelle, dei domini e degli altri oggetti del database** al fine di **definire il suo schema logico**.

Definizione delle tabelle Le tabelle vengono definite in SQL con l'istruzione **CREATE TABLE**. Questa istruzione **definisce uno schema di relazione**, ne **specifica attributi, domini e vincoli** e ne crea un'istanza vuota.

```
CREATE TABLE NomeTabella
        (NomeColonna1 TipoColonna1 ClausolaDefault1 VincoloColonna1,
         NomeColonna2 TipoColonna2 ClausolaDefault2 VincoloColonna2,
         ...
         NomeColonnak TipoColonnak ClausolaDefaultk VincoloColonnak,
         VincoliDiTabella
        )
```

CREATE TABLE è seguito dal nome della tabella e dalla **lista delle colonne** (attributi), di cui vengono specificate le caratteristiche. Alla fine si possono anche specificare eventuali vincoli di tabella, di cui parleremo.
La CREATE TABLE definisce uno schema di relazione e ne crea un'istanza vuota specificando attributi, domini e vincoli. Una volta create, la tabella è **pronta** per l'inserimento dei dati (che dovranno soddisfare i vincoli imposti).
Lo schema di una tabella, dopo che è stata creata, può essere visualizzato con **DESCRIBE NomeTabella**.

Un linguaggio per tanti usi SQL non è solo un query language, ma anche un linguaggio

per la definizione di DB (DDL)

```
CREATE SCHEMA Nome AUTHORIZATION Utente
CREATE TABLE
CREATE INDEX
CREATE PROCEDURE
CREATE TRIGGER
```

per stabilire controlli sull'uso dei dati

```
GRANT
```

per la modifica dei dati

4.2.1 Tipi

Tipi I tipi più comuni per i valori sono:

CHAR(n) per stringhe di caratteri di lunghezza fissa n

VARCHAR(n) per stringhe di caratteri di lunghezza variabile fino ad un massimo di n

INTEGER per interi con la dimensione uguale alla parola di memoria standard dell'elaboratore

REAL per numeri reali con dimensione uguale alla parola di memoria standard dell'elaboratore

NUMBER(p,s) per numeri con p cifre di cui s decimali

FLOAT(p) per numeri binari in virgola mobile con almeno p cifre significative

DATE per valori che rappresentano istanti nel tempo (in alcuni sistemi come Oracle) oppure solo date (con un altro tipo **TIME** per indicare l'orario)

```
CREATE TABLE Impiegati
       (Codice CHAR(8) NOT NULL,
        Nome CHAR(20),
        AnnoNascita INTEGER CHECK (AnnoNascita < 2000),
        Qualifica CHAR(20) DEFAULT 'Impiegato',
        Supervisore CHAR(8),
       PRIMARY KEY pk_impiegato (Codice),
       FOREIGN KEY fk_Impiegati (Supervisore) REFERENCES Impiegati
       )

CREATE TABLE FamiliariACarico
       (Nome CHAR(20),
        AnnoNascita INTEGER,
        GradoParentela CHAR(10),
        CapoFamiglia CHAR(8)
       FOREIGN KEY fk_FamiliariACarico (CapoFamiglia) REFERENCES Impiegati
       )
```

Eliminare e modificare Ciò che viene creato con ALTER TABLE può essere eliminato con **DROP** o modificato con **ALTER**. Con **ALTER TABLE** nello standard SQL è possibile

Aggiungere una colonna **ADD COLUMN**

Rimuovere una colonna **DROP COLUMN**

Modificare una colonna **MODIFY**

Aggiungere l'assegnazione di valori di default **SET DEFAULT** o eliminarli **DROP DEFAULT**

Aggiungere vincoli di tabella **ADD CONSTRAINT** o eliminarli **DROP CONSTRAINT**

Altre opzioni specifica dei linguaggi.

Capitolo 5

Viste

Viste logiche Le **view** possono essere definite come delle **tabelle virtuali**: i dati sono riaggregazioni dei dati contenuti nelle tabelle "fisiche" (unici veri contenitori dei dati). Le viste **non contengono fisicamente i dati ma forniscono una visione diversa, dinamicamente aggiornata, degli stessi dati delle tabelle fisiche**.

Appare all'utente come una normale tabella, in cui può effettuare interrogazioni e modifiche all'interno dei suoi privilegi.

Vantaggi

Semplificano la rappresentazione dei dati.
Oltre ad assegnare un nome ad una vista, la sintassi dell'istruzione **CREATE VIEW** consente di cambiare i nomi delle colonne.
In generale, uno dei requisiti per la progettazione di un DB è la **normalizzazione dei dati**. La forma normalizzata di un DB, sebbene permetta una corretta modellazione della realtà, a volte porta ad una **maggiore difficoltà di comprensione dei dati da parte dell'utente**. Le viste consentono quindi di fornire i dati all'utente in una forma più intuitiva. Consentono anche di **convertire le unità di misura e creare nuovi formati**.

Possono essere **convenienti per eseguire query molto complesse**.

Consentono di **proteggere il database**: le viste ad accesso limitato possono essere usate per **controllare le informazioni a cui accede un determinato utente** del database.

Indipendenza logica
Consentono di operare modifiche allo schema del database senza modificare le applicazioni che lo utilizzano se passano attraverso le viste. Un po' come le interfacce nell'OOP.

Limitazioni

Non consentono l'utilizzo degli operatori booleani UNION, INTERSECT ed EXCEPT.
Gli ultimi due possono essere realizzati mediante una semplice SELECT, ma lo stesso non si può dire di UNION.

Non è possibile usare ORDER BY.

Sintassi Il comando DDL che consente di definire una vista è

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

I nomi delle colonne nella ListaAttributi sono assegnati alle colonen della vista, che corrispondono **ordinatamente** alle colonne elencate nella SELECT della subquery.
Se questi non sono specificati, le colonne della vista assumono gli stessi nomi di quelli della/e tabella/e a cui si riferisce.
Di seguito un esempio

```
CREATE VIEW ImpiegatiAmmin (Matricola, Nome, Cognome, Stipendio)
AS      (SELECT Matri, Nome, Cognome, Stip
         FROM Impiegato
         WHERE Dipart = 'Amministrazione' AND Stipendio > 1000
        )
```

Nonostante il **contenuto** sia **dinamico**, la **struttura non lo è**. Se una vista è definita su una subquery che riferisce una tabella T a cui viene aggiunta una colonna, la definizione *non* viene estesa alla vista. Ossia la vista conterrà sempre le stesse colonne che aveva prima dell'inserimento della nuova colonna in T.

Vista di gruppo Una **vista di gruppo** è una vista in cui una delle colonne è una funzione di gruppo. Diventa obbligatorio assegnare un nome alla colonna della vista corrispondente alla funzione di gruppo. Un esempio:

```
CREATE VIEW A3 (CodFabbrica, NumVersioni)
AS      (SELECT CodFabbrica, SUM(NumVersioni)
         FROM Modelli
         GROUP BY CodBFabbrica
        )
```

È una vista di gruppo anche una vista definita in base ad una vista di gruppo

```
CREATE VIEW A4
AS      (SELECT NumVersioni
         FROM A3
        )
```

Eliminare le viste Le viste si eliminano con il comando

```
DROP NomeViste [RESTRICT | CASCADE]
```

RESTRICT: la vista viene eliminata solo se non è riferita nella definizione di altri oggetti.

CASCADE: la vista viene eliminata e vengono eliminate tutte le dipendenze da tale vista nelle altre definizioni dello schema.

Nell'esempio, **DROP VIEW A3 CASCADE** elimina anche A4 oltre ad A3, mentre
DROP VIEW A3 RESTRICT impedisce la cancellazione di A3 finché A4 è presente nello schema.

Viste modificabili Le viste si interrogano come le tabelle, ma **in generale non sono modificabili**. Per poterlo fare, **deve esistere una corrispondenza biunivoca fra le righe della vista e le righe di una tabella** di base ovvero:

SELECT senza DISTINCT e solo di sttributi

FROM una sola tabella modificabile

WHERE senza sottoquery

GROUP BY e HAVING non presenti nella definizione

Si potrebbe aggiornare direttamente le tabelle collegate, ma **ha senso aggiornare le view nel case di accesso dati controllato**.

CHECK OPTION L'opzione WITH CHECK OPTION messa alla fine della definizione della vista assicura che le operazioni di inserimento e modifica effettuate tramite la vista soddisfino la clausola WHERE della subquery.

Supponiamo che una vista V1 sia definita in termini di un'altra vista V2. Se V1 è creata con WITH CHECK OPTION, il **DBMS verifica che la nuova tupla t inserita soddisfi sia la definizione di V1 che quella di V2** (e di tutte le altre eventuali viste da cui V1 dipende), **indipendentemente dal fatto che V2 sia definita con** WITH CHECK OPTION.
Questo comportamento di default è equivalente a WITH **CASCADED** CHECK OPTION. Lo si può alterare definendo V1 WITH **LOCAL** CHECK OPTION, così facendo il DBMS verifica solo che t soddisfi la specifica di V1 e di tutte e sole le viste da cui V1 dipende per cui è stata specificata WITH CHECK OPTION.

Capitolo 6

Vincoli

6.1 Vincoli Intrarelazionali

Integrità I **vincoli di integrità** consentono di **limitare i valori ammissibili per una determinata colonna della tabella in base a specifici criteri**.
I **vincoli di integrità intrarelazionali**, ossia che non fanno riferimento ad altre relazioni, sono:

NOT NULL

UNIQUE, definisce chiavi
Può essere espresso in due forme:

Nella definizione di un attributo, se forma da solo la chiave.
Usato nella definizione dell'attributo, UNIQUE indica che non ci possono essere due valori uguali in quella colonna: è una chiave della relazione, ma non una chiave primaria.

Come elemento separato.
Può essere riferito anche a insiemi di attributi: ciò significa che non ci devono essere due righe per cui l'insieme dei valori corrispondenti agli attributi specificati siano uguali. In questo caso è dichiarato dopo la specifica delle colonne con UNIQUE (ListaAttributi)

PRIMARY KEY, chiave primaria. Una sola e implica il NOT NULL e l'UNIQUE.
Questo vincolo è simile a UNIQUE, ma definisce la chiave primaria della relazione ossia **un attributo che individua univocamente il dato**. Anche qua abbiamo due forme:

Nella definizione di un attributo, se forma da solo la chiave.

Come elemento separato, nel caso di chiave primaria formata da più attributi.

CHECK, vedremo più avanti

6.2 Vincoli Interrelazionali

Referenziali Sono quei vincoli che vengono imposti quando **gli attributi di due diverse tabelle devono essere messi in relazione**. Questo viene fatto per soddisfare l'esigenza di DB **non ridondanti** e per **mantenere i dati sincronizzati**.
Se due tabelle gestiscono gli stessi dati, **è bene che di essi non ce ne siano più copie** sia per non occupare troppa memoria sia per non modificare due volte lo stesso dato per mantenere la coerenza.

REFERENCES e FOREIGN KEY permettono di definire i **vincoli di integrità referenziale**. Anche qua due sintassi:

Per singoli attributi, come vincolo di colonna

```
Attributok REFERENCES TabellaEsterna(ColonnaRiferita)
```

Per insiemi di attributi, come vincolo di tabella

```
FOREIGN KEY (ColonneInterne) REFERENCES TabellaEsterna(ColonneRiferite)
```

6.2.1 Reazione alla violazione

Possiamo definire **politiche di reazione alla violazione**, ossia stabilire l'azione che il DBMS esegue quando si viola il vincolo. Questo può succedere quando si cancella o modifica una riga.
Tali reazioni vengono dichiarate al momento della definizione dei vincoli di foreign key rispettivamente mediante ON DELETE e ON UPDATE

```
Attributok REFERENCES TabellaEsterna(ColonnaRiferita)
        [ON DELETE | ON UPDATE Reazione]

FOREIGN KEY (ColonneInterne) REFERENCES TabellaEsterna(ColonneRiferite)
        [ON DELETE | ON UPDATE Reazione]
```

Reazioni alla DELETE

NO ACTION: impedisce il delete (default)

CASCADE: genera un delete a catena su tutte le righe dipendenti

SET NULL: assegna NULL ai valori della colonna che ha il vincolo referenziale

SET DEFAULT: assegna il valore di default ai valori della colonna che ha il vincolo referenziale

Reazioni alla UPDATE

NO ACTION: impedisce gli aggiornamenti che violano l'integrità referenziale (default)

CASCADE: i referenti vengono impostati al nuovo valore del riferito

SET NULL: i referenti vengono impostati a NULL

SET DEFAULT: i referenti vengono impostati al valore di default

6.3 CHECK

Un vincolo di **CHECK** richiede che una colonna o un insieme di colonne **soddisfi una condizione per ogni riga della tabella**. Il vincolo specificato deve essere un'espressione booleana che è valutata usando i valori della colonna che vengono inseriti o aggiornati nella riga.
Può essere espresso come vincolo di colonna se coinvolge un solo attributo, oppure come vincolo di tabella se coinvolge più attributi.

Capitolo 7

Trigger

Un **trigger** definisce un'**azione che il database deve attivare automaticamente quando si verifica un determinato evento nel database**. Possono essere usati per **migliorare l'integrità referenziale dichiarativa**, **imporre regole complesse** oppure **effettuare revisioni sulle modifiche** dei dati.

DML ma anche DDL L'**esecuzione è trasparente** all'utente e **vengono eseguiti automaticamente** quando specifici tipi di comandi (**eventi**) di manipolazione dei dati vengono eseguiti su specifiche tabelle.
Tali comandi comprendono i comandi DML INSERT, UPDATE e DELETE, ma gli ultimi DBMS prevedono trigger anche su alcune istruzioni DDL.
Anche gli aggiornamenti di specifiche colonne possono essere usati come trigger di eventi.

7.1 Struttura

I trigger si basano sul paradigma **Evento-Condizione-Azione** (**ECA**)

```
CREATE TRIGGER NomeTrigger
TipoDiTrigger Evento {, Evento}
ON TabellaTarget
[FOR EACH ROW]
[WHEN Condizione]
Azione
```

TipoDiTrigger: BEFORE o AFTER

Evento: INSERT/DELETE/UPDATE

Se si vuole specificare il trigger a livello di riga si mette anche FOR EACH ROW, altrimenti niente

Condizione che si deve verificare affinché il trigger venga eseguito

Azione, definita dal codice da eseguire se si verifica la condizione

Un esempio:

```
CREATE TRIGGER ControlloStipendio BEFORE INSERT ON Impiegati
DECLARE StipendioMedio FLOAT
BEGIN
        SELECT AVG(Stipendio) INTO StipendioMedio
        FROM Impiegati
        WHERE Dipartimento = :new.Dipartimento;
        IF :new.Stipendio > 2 * StipendioMedio THEN
                RAISE_APPL_ERR('Stipendio Alto')
        END IF;
END;
```

7.2 Tipi di trigger

7.2.1 Trigger a livello di riga

Vengono eseguiti **una volta per ogni riga modificata in una transazione**. Spesso usati in applicazioni di revisione dei dati e si rivelano utili per **operazioni di audit dei dati** e per **mantenere sincronizzati i dati distribuiti**.
Si creano con la clausola **FOR EACH ROW** nel CREATE TRIGGER.

7.2.2 Trigger a livello di istruzione

Vengono eseguiti **una volta per ciascuna transazione**, indipendentemente dal numero di righe che vengono modificate.
Vengono usati per **attività correlate ai dati** come per **imporre misure aggiuntive di sicurezza sui tipi di transazione che possono essere eseguiti su una tabella**.
Questo è il **tipo di trigger predefinito** del CREATE TRIGGER, non serve specificare clausole.

BEFORE/AFTER I trigger possono essere eseguiti prima o dopo l'utilizzo dei comandi, e all'interno e possibile riferire i vecchi e nuovi valori coinvolti nella transazione.
In caso di BEFORE, i valori **vecchi sono i valori attualmente nella tabella da modificare** e i **nuovi sono quelli che vogliamo inserire**.
In caso di AFTER, i valori **vecchi sono quelli che c'erano prima della modifica** mentre i **nuovi sono quelli presenti nella tabella alla fine della modifica**.

Attivi/Passivi Un trigger è **attivo quando modifica lo stato della base dei dati**, mentre è **passivo quando serve a provocare il fallimento della transazione sotto certe condizioni**.

Capitolo 8

Controllo degli accessi

Ogni componente dello schema può essere protetto: tabelle, attributi, viste, domini...
Il possessore della risorsa assegna dei privilegi agli altri utenti. Un utente predefinito (_system) rappresenta l'amministratore della base di dati ed ha completo accesso alle risorsa. Ogni privilegio è caratterizzato da:

la risorsa a cui si riferisce

l'utente che concede il privilegio

l'utente che riceve il privilegio

l'azione che viene permessa sulla risorsa

se il privilegio può essere trasmesso o meno ad altri utenti

Tipi di privilegi

SELECT: lettura dei dati

INSERT: inserire record

DELETE: cancellazione record

UPDATE: modifica record

REFERENCES: definire chiavi esterne

WITH GRANT OPTION: si possono trasferire i privilegi ad altri utenti

I privilegi si **garantiscono** con

GRANT (`Privilegi` | **ALL PRIVILEDGES**) **ON** `Oggetto` **TO** `Utenti` [**WITH GRANT OPTION**]

e si possono **revocare** (solo chi li ha garantiti) con

REVOKE [**GRANT OPTION FOR**] `Privilegi` **ON** `Oggetto` **FROM** `Utenti` [**CASCADE**]

La revoca di default è RESTRICT (non esegue il comando se la revoca dei privilegi all'utente comporti qualche altra revoca), CASCADE invece forza l'esecuzione. Quando si toglie un privilegio a U, lo si toglie anche a tutti coloro che lo hanno avuto solo da U (*attenzione agli effetti a catena*)

Capitolo 9

Programmazione

9.1 Uso di SQL da programmi

9.1.1 Problemi

Le problematiche principali che si incontrano quando si programmano applicazioni che comunicano con DB sono:

come **collegarsi al DB**

come **trattare gli operatori** SQL

come **trattare il risultato di un comando** SQL, cioè **le relazioni**

come **scambiare informazioni sull'esito delle operazioni**

9.1.2 Approcci

Linguaggio integrato (dati e DML) Linguaggio **disegnato ad-hoc per usare SQL**. I **comandi** SQL sono **controllati staticamente** dal traduttore ed **eseguiti dal DBMS**. Un esempio:

PL/SQL, Oracle

Un linguaggio integrato Un linguaggio per manipolare DB che integra DML (SQL) con il linguaggio ospite. **Linguaggio a blocchi** con una struttura del controllo completa che **contiene l'SQL come sottolinguaggio**. Permette di:

definire variabili di tipo scalare, record (annidato), insieme di scalari, insieme di record piatti, cursore

definire i tipi delle variabili a partire da quelli del DB

eseguire interrogazioni SQL ed esplorarne il risultato

modificare un DB

definire procedure e moduli

gestire il flusso del controllo, le transazioni, le eccezioni

Linguaggio convenzionale + API Linguaggio convenzionale che **usa delle funzioni di libreria per usare SQL**. I **comandi** SQL sono **stringhe passate come parametri alle funzioni**, che poi vengono **controllate dinamicamente** dal DBMS prima di eseguirle.
Quindi invece di modificare il compilatore di un linguaggio, si usa una libreria di funzioni/oggetti che operano su DB (API), alle quali si passa come parametro stringhe SQL e ritornano il risultato sul quale si opera con una logica ad iteratori. Qualche esempio:

Microsoft ODBC, C/C++ standard per Windows

Sun JDBC, l'equivalente Java

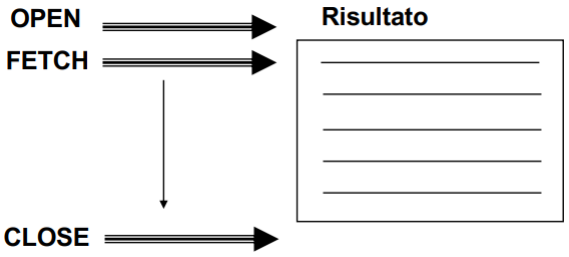
Dovrebbero essere **indipendenti dal DBMS**: un **driver** gestisce le richieste e le traduce in codice specifico per il singolo DBMS. Il DB può essere in rete.

Linguaggio che ospita l'SQL Linguaggio convenzionale **esteso con un nuovo costrutto per marcare i comandi** SQL. Occorre un **pre-compilatore** che **controlla i comandi SQL**, li **sostituisce con chiamate a funzioni predefinite** e **genera un programma in linguaggio convenzionale + API**.

Cursore Meccanismo per ottenere uno alla volta gli elementi di una relazione. Un cursore viene definito con un'espressione SQL poi:

si apre per far calcolare al DBMS il risultato, poi

con un opportuno comando si trasferiscono i campi delle ennuple in opportune variabili del programma



Capitolo 10

Normalizzazione

10.1 Teoria relazionale

Introduzione Ci sono due metodi per produrre uno schema relazionale:

Partire da un buon schema ad oggetti e tradurlo

Partire da uno schema relazionale già esistente e modificarlo o completarlo

Teoria della progettazione relazionale: si studia cosa sono le "**anomalie**" e **come eliminarle** (**normalizzazione**). Molto utile col secondo metodo di produzione, ma utile anche usando il primo.

Data una tabella, come si può dire se è fatta male, perché e come correggerla?

Esempio StudentiEdEsami(Matricola, Nome, Provincia, AnnoNascita, Materia, Voto)
Anomalie:

Ridondanze

Potenziali inconsistenze

Anomalie nelle **inserzioni**

Anomalie nelle **eliminazioni**

Prima soluzione: dividere lo schema in due tabelle

Studenti(Matricola, **Nome**, Provincia, AnnoNascita)

Esami(**Nome**, Materia, Voto)

Va bene? Potrebbero esserci omonimi, ancora meglio è

Studenti(**Matricola**, Nome, Provincia, AnnoNascita)

Esami(**Matricola**, Materia, Voto)

Dipendenze funzionali Nozione base. Obiettivi della teoria:

Equivalenza di schemi: in che misura si può dire che uno schema rappresenta un altro

Qualità degli schemi (**forme normali**)

Trasformazione degli schemi (**normalizzazione** degli schemi)

Ipotesi dello **schema di relazione universale**: tutti i fatti sono descritti da attributi di un'unica relazione (**relazione universale**), cioè gli **attributi hanno un significato globale**.

Definizione Lo **schema di relazione universale U** di un DB relazione **ha come attributi l'unione degli attributi di tutte le relazioni della base di dati**.

10.2 Forme normali

Una **forma normale** è una **proprietà di un DB relazionale che ne garantisce la qualità**, cioè l'assenza di determinati difetti.
Quando una relazione non è normalizzata **presenta ridondanze** o **si presta a comportamenti poco desiderabili** durante gli aggiornamenti.
La **normalizzazione** è una procedura che permette di trasformare schemi non normalizzati in schemi che soddisfano una forma normale.

Perché questi fenomeni indesiderabili?

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

Ridondanza: lo stipendio di ogni impiegato è ripetuto in tutte le ennuple relative per ogni progetto a cui partecipa, questo perché **Stipendio *dipende* solo da Impiegato**

Anomalia da aggiornamento: se lo stipendio varia, è necessario andarne a modificare il valore in diverse ennuple

Anomalia da cancellazione: se un impiegato interrompe la partecipazione a tutti i progetti, dobbiamo cancellare tutte le ennuple in cui appare e così l'impiegato non è più presente nel DB

Anomalia da inserimento: un nuovo impiegato non può essere inserito finché non è assegnato ad un progetto

10.2.1 Linee guida per una corretta progettazione

Semantica degli attributi Si progetti ogni schema relazionale in modo che **sia semplice spiegarne il significato**. Non si uniscano attributi provenienti da più tipi di classi e tipi di associazione in un unica relazione.

Ridondanza Si progettino gli schemi relazionali in modo che nelle relazioni **non siano presenti anomalie** di inserimento, cancellazione o modifica. Se sono presenti anomalie, le si rilevi chiaramente e ci si assicuri che i programmi che aggiornano il DB operino correttamente.

Valori nulli Per quanto possibile, **si eviti di porre attributi i cui valori possono essere frequentemente nulli**. Se è inevitabile, ci si assicuri che essi si presentino solo in casi eccezionali e che non riguardino una maggioranza di tuple nella relazione.

Tuple spurie Si progettino schemi in modo tale che essi possano essere riuniti tramite join con condizioni di uguaglianza su attributi che sono chiavi primarie o chiavi esterne, in modo da garantire che non vengano generate tuple spurie. **Non si abbiano relazioni con attributi di accoppiamento diversi dalle combinazioni chiave esterna-chiave primaria**.

10.2.2 Dipendenze funzionali

Per formalizzare la nozione di schema senza anomalie, occorre descrivere formalmente della semantica dei fatti rappresentati in uno schema relazionale.
Istanza valida di R: è una **nozione semantica** che dipende da ciò che sappiamo sul dominio del discorso (non estensionale né deducibile da alcune istanze dello schema → confronto con il committente!).
Nozione fondamentale: dipendenza funzionale.

Dipendenza funzionale Data un'istanza valida r su R(T). Siano X e Y due sottoinsiemi non vuoti di T.
$\exists$ in r una **dipendenza funzionale** da X a Y se, $\forall$ coppia di ennuple t_1, t_2 di r con gli stessi valori su X risulta che t_1 e t_2 hanno gli stessi valori anche su Y.
La dipendenza funzionale da X a Y si denota con X $\rightarrow$ Y.

Esempio: Persone(CodiceFiscale, Cognome, Nome, DataNascita) ha CodiceFiscale $\rightarrow$ Cognome.
Più formalmente:

Dato uno schema R(T) e X, Y $\subseteq$ T, una **dipendenza funzionale** (DF) fra gli attributi X e Y è un vincolo su R sulle istanze della relazione espresso nella forma X $\rightarrow$ Y, che significa **X determina funzionalmente Y** o **Y è determinato da X**, se $\forall$ r istanza valida di R $\Rightarrow$ $\forall$ $t_1, t_2 \in r$ si ha $(t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y])$

Esempio

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

Impiegato $\rightarrow$ Stipendio
Progetto $\rightarrow$ Bilancio
Impiegato Progetto $\rightarrow$ Funzione

Le **dipendenze funzionali sono una proprietà semantica**, cioè **dipendono dai fatti rappresentati** e non dalla rappresentazione degli attributi negli schemi.

Notazione Con R$\langle$T, F$\rangle$ si denota uno schema con attributi T e dipendenze funzionali F.
Si parla di dipendenza funzionale **completa** quando X $\rightarrow$ Y e $\forall$ W $\subset$ X non vale W $\rightarrow$ Y.
Se X è una superchiave, allora X determina ogni altro attributo della relazione cioè X $\rightarrow$ T. Se X è chiave, allora X $\rightarrow$ T è una dipendenza funzionale completa.

Proprietà Da un insieme F di dipendenze funzionali, in generale altre dipendenze funzionali sono "implicate" da F.
Per esempio (Matricola $\rightarrow$ CodFisc, CodFisc $\rightarrow$ Cognome allora Matricola $\rightarrow$ Cognome)

> **Dipendenze implicate**: sia F un insieme di DF sullo schema R, diremo che F **implica logicamente** X $\rightarrow$ Y (F $\vDash$ X $\rightarrow$ Y) se ogni istanza r di R che soddisfa F soddisfa anche X $\rightarrow$ Y
>
> **Dipendenze banali**: implicate dal vuoto.
> Ad esempio $\{\} \vDash X \rightarrow Y$

Come derivare le DF implicate logicamente da F? Usando un **insieme di regole d'inferenza**.

Assiomi di Armstrong Sono in realtà regole d'inferenza:

> **Riflessività R**: Y $\subseteq$ X $\Rightarrow$ X $\rightarrow$ Y
>
> **Arricchimento A**: X $\rightarrow$ Y $\wedge$ Z $\subseteq$ T $\Rightarrow$ XZ $\rightarrow$ YZ
>
> **Transitività T**: X $\rightarrow$ Y $\wedge$ Y $\rightarrow$ Z $\Rightarrow$ X $\rightarrow$ Z

Derivazione Sia F un insieme di DF. Diremo che **$X \to Y$ sia derivabile da F ($F \vdash X \to Y$)** se $X \to Y$ può essere inferito da F usando gli assiomi di Armstrong.
Si dimostra che valgono anche le seguenti regole:

Unione U: $\{X \to Y, X \to Z\} \vdash X \to YZ$

Decomposizione D: $\{Z \subseteq Y, X \to Y\} \vdash X \to Z$

Una derivazione di f da F è una sequenza finita $f_1, \ldots, f_m$ di dipendenze, dove $f_m = f$ e ogni f_i è un elemento di F oppure è ottenuta dalle precedenti dipendenze $f_1 \ldots, f_{i-1}$ della derivazione usando una regola d'inferenza.

Correttezza e completezza Gli assiomi di Armstrong sono corretti e completi. Attraverso essi, si può dimostrare l'equivalenza della nozione di implicazione logica ($\vDash$) e di quella di derivazione ($\vdash$): **se una dipendenza è derivabile con gli assiomi di Armstrong, allora è anche implicata logicamente** (correttezza) e viceversa **se una dipendenza è implicata logicamente allora è anche derivabile dagli assiomi** (completezza).
Formalmente:

$\forall f \quad F \vdash f \Rightarrow F \vDash f$

$\forall f \quad F \vDash f \Rightarrow F \vdash f$

Chiusura di un insieme F Dato un insieme F di DF, la **chiusura di F**, denotata con F^+, è
$F^+ = \{X \to Y \mid F \vdash X \to Y\}$
Si presenta spesso il problema di decidere se una dipendenza funzionale appartiene a F^+ (**problema dell'implicazione**). La sua risoluzione con l'algoritmo banale (di generare F^+ applicando ad F ripetutamente gli assiomi di Armstrong) ha una **complessità esponenziale** rispetto al numero di attributi dello schema.

Dato $R\langle T, F\rangle$ e $X \subseteq T$, la **chiusura di X rispetto ad F**, denotata con X_F^+ (o X^+ se F è chiaro dal contesto) è $X_F^+ = \{A_i \in T \mid F \vdash X \to A_i\}$

Problema dell'implicazione Controllare se una DF $V \to W \in F^+$.
Un algoritmo efficiente per risolverlo senza calcolare F^+ parte dal seguente **teorema**: $F \vdash X \to Y \Leftrightarrow Y \subseteq X_F^+$

Idea Sia X un insieme di attributi e F un insieme di dipendenze. Vogliamo calcolare X_F^+.

1. Inizializziamo X^+ con l'insieme X
2. Se fra le dipendenze di F c'è una dipendenza $Y \to A$ con $Y \subseteq X^+$ allora si inserisce A in X^+ ossia $X^+ = X^+ \cup \{A\}$
3. Si ripete 2. fino a quando non ci sono altri attributi da aggiungere a X^+
4. Output $X_F^+ = X^+$

Chiavi e attributi primi Dato lo schema $R\langle T, F\rangle$, diremo che un insieme di attributi $W \subseteq T$ è una **chiave candidata** di R se

W **superchiave**, cioè $W \to T \in F^+$

$V \subset W \Rightarrow V$ non superchiave, cioè $\forall\, V \subset W, V \to T \notin F^+$

Attributo primo: attributo che appartiene ad almeno una chiave.

Il problema di trovare tutte le chiavi di una relazione richiede un algoritmo di complessità esponenziale nel caso peggiore.
Il problema di controllare se un attributo è primo è NP-Completo.

Trovare tutte le chiavi Per trovare tutte le chiavi di R⟨T,F⟩ si parte da un insieme di candidati che hanno come base pari all'insieme T meno tutte le parti destre delle dipendenze funzionali in F. Ogni candidato è un sottoinsieme di T rappresentato come X::(Y), che denota dato Y = {A_1, ..., A_n} tutti gli insiemi formati da X e da un qualsiasi insieme di attributi A_i.
Quindi se Base sono gli attributi che non appaiono a destra di nessuna dipendenza, tali attributi devono apparire in ogni chiave per cui inizialmente i candidati sono Base::(T - Base).

Ogni insieme in X::(Y) è analizzato partendo da X: se è già chiave, allora tutti gli altri insiemi X::(Y) sono scartati, altrimenti si mettono in candidati XA_1::(Y-A_1), ..., XA_n::(Y-A_n).
Se X non contiene chiavi già trovate in precedenza, e $X^+ = T$, allora X è chiave.

Le chiavi trovate dopo saranno per forza più lunghe, e non potranno essere contenute in una chiave già trovata. Questo si assicura aggiungendo i nuovi candidati in coda alla lista (append), mantenendo quindi la lista dei candidati ordinata per lunghezza.

Esempio Con T = {A, B, C, D, E, F} e F = {C→D, CF→B, D→C, F→E}
La Base è inizializzata quindi con AF attributi, quindi i candidati sono AF::(BCDE).
Testando AF, la sua chiusura $AF^+ = \{A, F, E\} \neq T$, quindi si prosegue. I candidati sono quindi AF::(BCD) - AF:

$AFB^+ = \{A, F, B, E\} \neq T$

$AFC^+ = \{A, F, C, E, D, B\} = T$ è chiave

$AFD^+ = \{A, F, D, C, E, B\} = T$ è chiave

AFBC non si testa, AFC è chiave

AFBD non si testa, AFD è chiave

10.3 Copertura Canonica

Copertura Due insiemi di dipendenze funzionali F e G sullo schema R sono **equivalenti** $F \equiv G \Leftrightarrow F^+ = G^+$
$F \equiv G \Rightarrow$ F è una **copertura** di G (e G è una copertura di F).

Attributo estraneo Sia F un insieme di DF.
Data una $X \to Y \in F$, si dice che X **contiene un attributo estraneo** $A_i \Leftrightarrow (X - \{A_i\}) \to Y \in F^+$, cioè
$F \vdash (X - \{A_i\}) \to Y$
Per verificare se A è estraneo in $AX \to B$, calcoliamo X^+ e verifichiamo se include B, cioè se basta X a determinare B.

Un esempio Consideriamo Orari(CodAula, NomeAula, Piano, Posti, Materia, CDL, Docente, Giorno, Ora)
Se vale

Docente, Giorno, *Ora* → CodAula

Docente, Giorno → *Ora*

Allora

Docente, Giorno → CodAula

Nella prima dipendenza ***Ora* è attributo estraneo**.

Dipendenza ridondante Sia F un insieme di DF.
$X \to Y$ è una **dipendenza ridondante** $\Leftrightarrow (F - \{X \to Y\})^+ = F^+$, cioè $F - \{X \to Y\} \vdash X \to Y$
Per verificare se $X \to A$ è ridondante la eliminiamo da F, calcoliamo X^+ e verifichiamo se include A, ovvero se con la DF che restano riusciamo ancora a dimostrare che X determina A.

Esempi

F_1={A → B, AB → C, A → C}
Ridondante perché {A → B, AB → C} ⇒ A → C

F_2={A → B, AB → C}
Non è ridondante, ma B è estraneo perché può essere eliminato dal primo membro della seconda dipendenza

F_3={A → B, A → C}
Non presenta attributi estranei

Copertura canonica F è detta copertura canonica ⇔

la parte destra di ogni DF ∈ F è un attributo

non esistono attributi estranei

nessuna dipendenza è ridondante

Teorema: per ogni insieme F di DF esiste una copertura canonica.
La copertura canonica si calcola così:

Trasformare le dipendenze nella forma X → A
Si sostituisce l'insieme dato con quello equivalente che ha tutti i secondi membri costituiti da singoli attributi (**dipendenze atomiche**)

Eliminare gli attributi estranei
Per ogni dipendenza si verifica se esistono attributi eliminabili dal primo membro. Cioè ∀ X → A ∈ F, verifichiamo se esiste Y ⊆ X | F ≡ F - {X → A} ∪ {Y → A}

Eliminare le dipendenze ridondanti

10.4 Decomposizione di Schemi

In generale, per eliminare le anomalie da uno schema, occorre decomporlo in schemi più piccoli "equivalenti".

Definizione Dato uno schema R(T), $\rho = \{R_1(T_1),\ldots, R_k(T_k)\}$ è una **decomposizione di R** ⇔ $T_1 \cup \ldots \cup T_k = T$
Una decomposizione deve avere due **proprietà desiderabili**:

Conservazione dei dati (nozione semantica)
Una decomposizione $\rho = \{R_1(T_1),\ldots, R_k(T_k)\}$ di uno schema R(T) **preserva i dati** ⇔ ∀ istanza **valida** r di R si ha $r = (\pi_{T_1} r) \vee \ldots \vee (\pi_{T_k} r)$
Dalla definizione di join naturale si ottiene un **teorema**: se $\rho = \{R_1(T_1),\ldots, R_k(T_k)\}$ è una decomposizione su R(T) ⇒ ∀ istanza r di R si ha $r \subseteq (\pi_{T_1} r) \vee \ldots \vee (\pi_{T_k} r)$
Uno schema R(T) si **decompone senza perdita** negli schemi $R_1(T_1)$ ed $R_2(T_2)$ se, per ogni possibile istanza r di R(T), il **join naturale delle proiezioni di r su T_1 ed T_2 produce la tabella di partenza** (cioè senza ennuple spurie).
Cioè $\pi_{T_1}(r) \bowtie \pi_{T_2}(r) = r$. La decomposizione senza perdita è garantita se l'insieme degli attributi comuni alle due relazioni ($T_1 \cap T_2$) è **chiave per almeno una delle due relazioni decomposte**. Questo perché:

Supponiamo r relazione sugli attributi ABC, consideriamo le sue proiezioni r_1 su AB e r_2 su AC.

Supponiamo r soddisfi la DF A → C, allora **A è chiave per r_1 su AC**, quindi non ci sono in tale proiezione due tuple diverse sui gli stessi valori di A.

Il join costruisce tuple a partire dalle tuple nelle proiezioni.

Sia $t = (a, b, c)$ una tupla del join di r_1 ed r_2, mostriamo che appartiene ad r (cioè non è spuria):

t è ottenuta tramite join da $t_1 = (a, b)$ di r_1 e $t_2 = (a, c)$ su r_2

Allora per definizione di proiezione esistono due tuple in r $t'_1 = (a, b, *)$ e $t'_2 = (a, *, c)$, dove $*$ sta per valore non noto.

Dato che A → C in r allora ∃ un solo valore in C associato al valore a. Dato che (a, c) compare nella proiezione, questo valore è proprio c

Ma allora in t'_1 ci deve essere proprio c, quindi $(a, b, c) \in r$

Conservazione delle dipendenze
Dato lo schema R⟨T, F⟩, la decomposizione $\rho = \{R_1, \ldots, R_n\}$ **preserva le dipendenze** ⇔ l'**unione delle dipendenze in** $\pi_{T_i}(\mathbf{F})$ **è una copertura di F**.
Il problema di stabilire se la decomposizione $\rho = \{R_1, \ldots, R_n\}$ preserva le dipendenze ha complessità di tempo polinomiale. Un **teorema** importante dice che dato $\rho = \{R_i\langle T_i, F_i\rangle\}$ una decomposizione di R⟨T, F⟩ che **preserva le dipendenze** e tale che T_j **sia una superchiave di R**, allora ρ **preserva i dati**.

Qualità Una decomposizione dovrebbe sempre soddisfare le seguenti **proprietà**:

la **decomposizione senza perdita**, che garantisce la ricostruzione delle informazioni originarie senza generazione di tuple spurie

la **conservazione delle dipendenze**, che garantisce il mantenimento dei vincoli d'integrità originari

Soddisfacimento della FNBC: ogni tabella prodotta deve essere in FNBC

10.5 Forme Normali

Una **forma normale** è una **proprietà** di un DB relazionale che ne **garantisce la qualità**, cioè l'assenza di determinati effetti. Quando una relazione non è normalizzata presenta ridondanze e si presta a comportamenti poco desiderabili durante gli aggiornamenti. Ci sono vari tipi di forme normali:

1FN: impone una restrizione sul tipo di una relazione, in cui **ogni attributo ha un tipo elementare**

2FN, 3FN e FNBC: impongono **restrizioni sulle dipendenze**. FNBC/BCNF (Boyce-Codd) è la più naturale e **restrittiva**.

10.5.1 FNBC

Una relazione r è in FNBC se ∀ DF non banale X → Y definita su di essa, X **contiene una chiave K di** r (cioè è superchiave).
La forma normale non richiede che i concetti in una relazione siano omogenei (solo proprietà direttamente associate alla chiave). L'intuizione è che se esiste in R una dipendenza X → A non banale ed X non è chiave, allora X modella l'identità di un'entità diversa da quelle modellate dall'intera R.

Definizione R⟨T,F⟩ è in BCNF ⇔ ∀ X → A ∈ F^+, con A ∉ X (non banale), X è una superchiave

Teorema R⟨T,F⟩ è in BCNF ⇔ ∀ X → A ∈ F non banale, X è una superchiave

L'algoritmo di analisi R⟨T,F⟩ è decomposta in R_1(X, Y) e R_2(X, Z) e su di esse si ripete il procedimento, esponenziale.

A vole la decomposizione per raggiungere la BCNF può risultare difficile o impossibile, ed occorre quindi ricorrere ad una forma normale indebolita.

10.5.2 3FN

Una relazione r è in 3NF se ∀ DF non banale X → Y definita su r, è verificata almeno una delle seguenti condizioni:

X contiene una chiave K di r (come nella BCNF)

ogni attributo in Y è contenuto in almeno una chiave K di r

Meno restrittiva della FNBC: tollera alcune ridondanze e certifica meno la qualità dello schema ottenuto, ma è **sempre ottenibile** qualunque sia lo schema di partenza. Basta usare l'algoritmo di normalizzazione in TFN.

10.5.3 Algoritmo di Sintesi (versione base)

Input Insieme R di attributi e insieme F di dipendenze su R

Output Decomposizione $\rho = \{S_i\}_{i=1...n}$ di R tale che preservi i dati e le dipendenze ed ogni S_i sia in 3NF rispetto alle proiezioni di F su S_i.

1. Trova una copertura canonica G di F e poni $\rho = \{\}$
2. Sostituisci in G ogni insieme $X \to A_1, \ldots, X \to A_h$ di dipendenze con lo stesso determinante con la dipendenza $X \to A_1 \ldots A_h$
3. Per ogni dipendenza $X \to Y$ in G metti uno schema con attributi XY in ρ
4. Elimina da ρ ogni schema che sia contenuto in un altro schema di ρ
5. Se la decomposizione non contiene alcuno schema i cui attributi costituiscano una superchiave per R, aggiungi ad essa lo schema con attributi W, con W una chiave di R.

Capitolo 11

DBMS

Un DBMS deve gestire grandi quantità di dati persistenti e condivisi. La gestione richiede **particolare attenzione ai problemi di efficienza**, come l'ottimizzazione delle richieste ma non solo.
La **persistenza** e la **condivisione** richiedono che un DBMS fornisca dei **meccanismi per garantire l'affidabilità dei dati** (**fault tolerance**), per il **controllo degli accessi** e per il **controllo della concorrenza**. Diverse altre funzionalità vengono messe a disposizione per motivi di efficacia, ovvero per semplificare la descrizione dei dati, lo sviluppo delle applicazioni, l'amministrazione del DB...

Condivisione dei dati La **gestione integrata** e la **condivisione dei dati** permettono di evitare le ripetizioni (ridondanza dovuta a copie multiple dello stesso dato) e quindi di **evitare un inutile spreco di risorse** (memoria). Inoltre, la **ridondanza può dare luogo a problemi di inconsistenza** delle copie e comporta la necessità di propagare le modifiche, con un ulteriore spreco di risorse (CPU e rete).

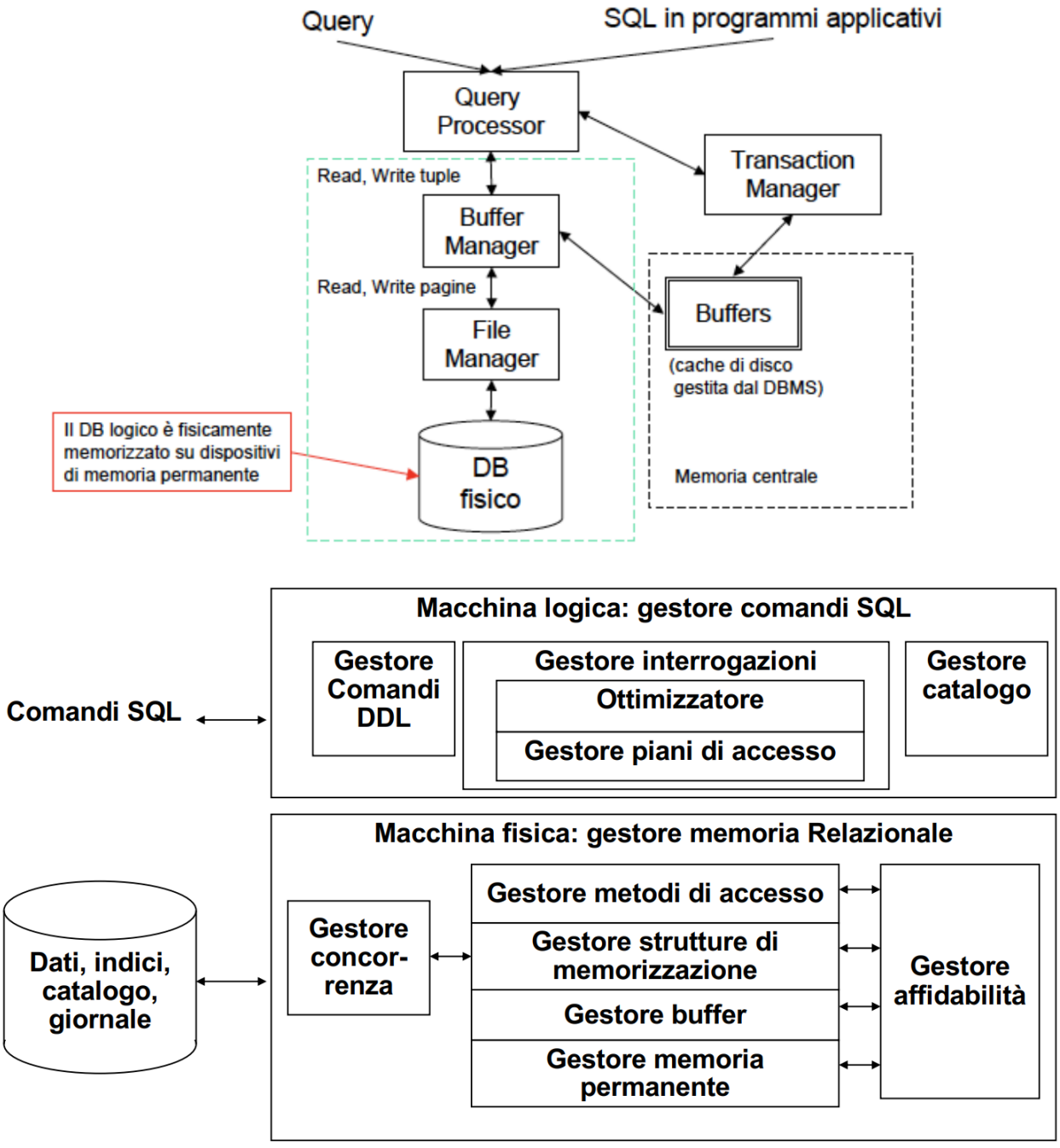
Modello dei dati Dal punto di vista utente, un DB è visto come una collezione di dati che modellano una certa porzione della realtà di interesse.
L'**astrazione logica** con cui i dati vengono resi disponibili all'utente **definisce un modello dei dati**. Più precisamente: **un modello dei dati è una collezione di concetti che vengono utilizzati per descrivere i dati, le loro associazioni/relazioni ed i vincoli che questi devono rispettare**.
Un **ruolo di primaria importanza** nella definizione di un modello dei dati è svolto dai **meccanismi che possono essere usati per strutturare i dati** (tipo i costruttori di tipo in un linguaggio di programmazione). Per esempio, ci sono modelli in cui i dati sono descritti solo sotto forma di albero (modello **gerarchico**), di grafi (modello **reticolare**), di oggetti complessi (modello **a oggetti**), di relazioni (modello **relazionale**).

Indipendenza fisica e logica Tra gli obiettivi di un DBMS, vi sono quelli di fornire caratteristiche di

> **Indipendenza fisica** dei dati: dipende da considerazioni legate all'efficienza delle organizzazioni adottate. **La riorganizzazione fisica dei dati non deve comportare effetti collaterali sui programmi applicativi**.

> **Indipendenza logica**: permette di **accedere ai dati logici indipendentemente dalla loro rappresentazione fisica**.

11.1 Architettura semplificata di un DBMS



Gerarchia di memoria La memoria in un sistema di calcolo è organizzata in una gerarchia: al livello più alto ci sono memorie di piccola dimensione, molto veloci e costose, mentre più si scende più la dimensione aumenta e si riducono velocità e costi.
Le prestazioni si misurano in termini di tempo di accesso

$$\textit{Tempo di accesso} = \textit{latenza} + \frac{\textit{dimensione dati da trasferire}}{\textit{velocità di trasferimento}}$$

A causa delle dimensioni, un DB solitamente risiede sui dischi. I dati devono essere trasferiti in memoria centrale per essere elaborati dal DBMS: il **trasferimento non avviene a singole tuple ma a blocchi**, anche detti pagine.

Spesso le operazioni di I/O costituiscono il **collo di bottiglia di un sistema**, quindi è necessario

ottimizzare l'implementazione fisica del DB attraverso:

Organizzazione efficiente delle tuple su disco

Strutture di accesso efficienti

Gestione efficiente dei buffer in memoria

Strategie di esecuzione efficienti per le query

11.1.1 Gestore di memoria permanente

Fornisce un'astrazione della memoria permanente in termini di insiemi di file logici, di pagine fisiche e di blocchi, nascondendo le caratteristiche dei dischi e del sistema operativo.

11.1.2 Gestore del buffer

Si preoccupa del trasferimento delle pagine tra la memoria temporanea e la memoria permanente, offrendo agli altri livelli una visione della memoria permanente come un insieme di pagine utilizzabili in memoria temporanea, astraendo da quando esse vengano trasferite dalla memoria permanente al buffer e viceversa.
Nei DBMS, la LRU per rimpiazzare le pagine non è sempre una buona scelta. Per alcune query, il "pattern di accesso" ai dati è noto e quindi può essere utilizzato per operare scelte più accurate, in grado di migliorare anche di molto le prestazioni.

11.1.3 Gestore strutture di memorizzazione

Tipi di organizzazione

Seriali/Sequenziali
Organizzazione **seriale** (**heap file**): i dati sono memorizzati in modo **disordinato** uno dopo l'altro. Semplice, a basso costo di memoria, poco efficiente quindi va bene per pochi dati. Organizzazione standard di ogni DBMS.
Organizzazione **sequenziale**: i dati sono **ordinati** sul valore di uno o più attributi, così da avere ricerche più veloci, ma le nuove inserzioni fanno perdere l'ordinamento. Costo di ricerca di $\log_2 b_i + 1$ accessi per ogni blocco, se ogni file contiene b_i blocchi dato che la ricerca binaria richiede $\log_2 b$ accessi.-

Per chiave
Obiettivo: noto il valore di una chiave, trovare il record di una tabella con qualche accesso al disco (idealmente uno solo).
Alternative: metodo procedurale (hash) o tabellare (indice), organizzazione statica o dinamica.

Per attributi non chiave

Parametri che caratterizzano un'organizzazione

Occupazione di memoria

Costo delle operazioni di: ricerca, modifica, inserzione, cancellazione

Ordinamento di archivi Ordinare gli archivi è importante per eseguire alcune operazioni relazionali (JOIN, SELECT DISTINCT, GROUP BY) e per avere risultati ordinati dalle interrogazioni (ORDER BY).

Algoritmo L'algoritmo utilizzato costa N·log(N): è lo Z-way merge sort (merge sort a Z vie).
Supponiamo di dover ordinare un input che consiste in un file di NP pagine e di avere a disposizione solo NB < NP buffer in memoria centrale. L'algoritmo opera in due fasi:

Sort interno: si leggono una alla volta le pagine del file. I **record di ogni pagina sono ordinati facendo uso di un algoritmo di sort interno** (es: quicksort). **Ogni pagina così ordinata**, detta anche **run**, **viene scritta su disco** in un file **temporaneo**.

Merge: operando **uno o più passi di fusione** le **run vengono fuse fino a produrne una sola**.

Per la **complessità**, consideriamo il solo numero di operazioni I/O al caso base Z = 2 e NB = 3. Si osserva che:

Nella fase di sort interno si leggono e riscrivono NP pagine

Ad ogni passo di merge si leggono e riscrivono NP pagine
Il numero di passi di fusione è pari a $\lceil \log_2 NP \rceil$, in quanto ad ogni passo il numero di run si dimezza.

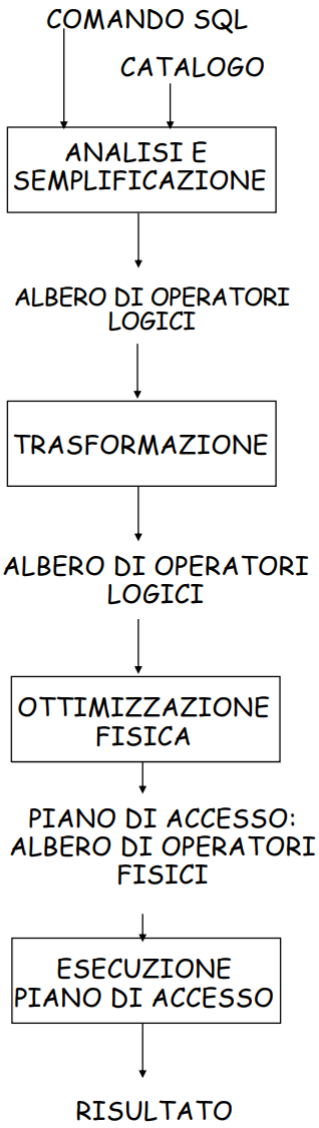
Il **costo complessivo** è pertanto $2 \cdot NP \cdot (\lceil \log_2 NP \rceil + 1)$

Al caso generale, possiamo osservare che nel passo di sort interno si possono ordinare NB pagine alla volta invece che una sola, il che abbassa il costo a $2 \cdot NP \cdot (\lceil \log_2 (NP/NB) \rceil + 1)$

Oltre che per ordinare le tuple, il sort può essere usato per le query dove compare DISTINCT, per eliminare i duplicati, oppure anche per le query contenenti GROUP BY.

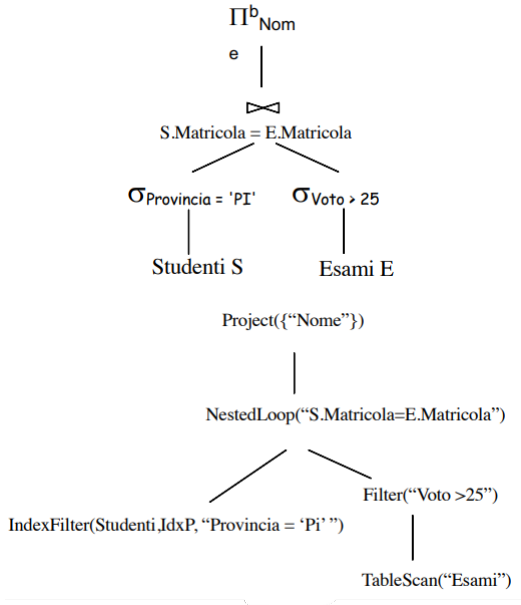
11.2 Piani di accesso

1. Analisi lessicale e sintattica del comando SQL Q
   `SQLCommand parseTree = Parser.parseStatement(Q);`
2. Analisi semantica del comando
   `Type type = parseTree.check();`
3. Ottimizzazione dell'interrogazione
   `Value pianoDiAccesso = parseTree.Optimize();`
4. Esecuzione del piano di accesso



Analisi e semplificazione: verifica la correttezza del comando, normalizzazione e semplificazione della condizione.
Piano di accesso: **scelta** dell'algoritmo per eseguire ogni operatore: **Ideale**, trovare il piano migliore, ed **Euristica**, evitare i piani peggiori.

```
SELECT  Nome
FROM    Studenti S, Esami E
WHERE   S.Matricola = E.Matricola
        AND Provincia = "PI"
        AND VOTO > 25
```



Le foglie sono le tabelle ed i nodi interni specificano le modalità con cui gli accessi alle tabelle e le operazioni relazionali sono effettuate.

Realizzazione degli operatori relazionali Si considerano gli operatori di: proiezione, selezione, raggruppamento, join.
Un **operatore può essere realizzato con algoritmi diversi**, **codificati in opportuni operatori fisici**.

Operatori fisici Gli algoritmi per realizzare gli operatori relazionali si codificano in opportuni operatori fisici. Ad esempio, `TableScan(R)` è l'operatore fisico per la scansione di R.
Ogni operatore fisico è un **iteratore**, cioè un **oggetto con metodi** `open`, `next`, `isDone`, `reset` e `close`, realizzati usando gli operatori della macchina fisica e con `next` che ritorna un record.
Prenderemo come esempio gli operatori fisici del sistema JRS, usandoli per descrivere un algoritmo che esegue un'interrogazione SQL.

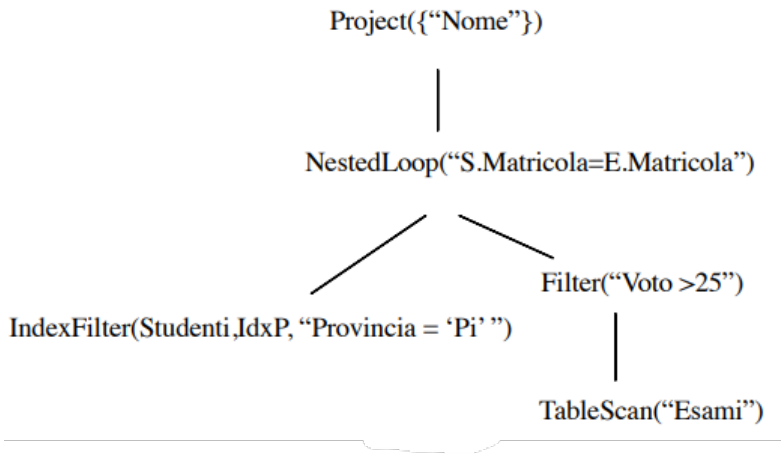
`open`: inizializza lo stato dell'operatore, alloca buffer per gli input e l'output, richiama ricorsivamente open sugli operatori figli; viene anche usato per passare argomenti (ad es. la condizione che un operatore `Filter` deve applicare)

`next`: usato per richiedere un'altra tupla del risultato dell'operatore. L'implementazione di questo metodo include `next` sugli operatori figli e codice specifico dell'operatore

`close`: usato per terminare l'esecuzione dell'operatore, con conseguente rilascio delle risorse ad esso allocate

`isDone`: indica se vi sono ancora valori da leggere, in genera è booleano.

Il **piano di accesso** quindi, cioè l'algoritmo che viene usato per eseguire un'interrogazione usando gli operatori fisici disponibili, è rappresentato da un albero del tipo



Operatore logico	Operatore fisico
R	`TableScan(R)` per la scansione di R `IndexScan(R, Idx)` per la scansione di R con l'indice Idx `SortScan(R, {`A_i`})` per la scansione di R ordinata sugli $\{A_i\}$
$\pi^b_{\{A_i\}}$ $\pi_{\{A_i\}}$	`Project(O, {`A_i`})` per la proiezione dei record i O senza l'eliminazione dei duplicati `Distinct(O)` per eliminare i duplicati dei record ordinati da O
σ_ψ	`Filter(O,` ψ`)` per la restrizione senza indici dei record di O `IndexFilter(R, Idx,` ψ`)` per la restrizione con indice dei record di R
$\tau_{\{A_i\}}$	`Sort(O, {`A_i`})` per ordinare i record di O sugli $\{A_i\}$, per valori crescenti
${}_{\{A_i\}}\gamma_{\{f_i\}}$	`GroupBy(O, {`A_i`}, {`f_i`})` per raggruppare i record di O sugli $\{A_i\}$ usando le funzioni di aggregazione in $\{f_i\}$. Nell'insieme $\{f_i\}$ vi sono le funzioni di aggregazione presenti nella SELECT e nella HAVING. L'operatore ritorna record con attributi gli $\{A_i\}$ e le funzioni in $\{f_i\}$ I record di O sono ordinati sugli $\{A_i\}$
$\bowtie_{\psi_j}$	`NestedLoop(`O_E`, `O_I`,` ψ_j`)` per la join con il **nested loop** e ψ_j condizione di join `PageNestedLoop(`O_E`, `O_I`,` ψ_j`)` per la join con il **page nested loop** `IndexNestedLoop(`O_E`, `O_I`,` ψ_j`)` per la join con l'**index nested loop**. L'operando interno O_I è un `IndexFilter(R, Idx,` ψ_j`)` oppure `Filter(O,` ψ'`)` con O un `IndexFilter(R, Idx,` ψ_j`)`. Per ogni record r di O_E, la condizione ψ_j dell'`IndexFilter` è quella di giunzione con gli attributi di O_E sostituiti dai valori in r. `SortMerge(`O_E`, `O_I`,` ψ_j`)` per la giunzione con il **sort-merge**, con i record di O_E e O_I ordinati sugli attributi di giunzione

11.3 Transazioni

Durante le transazioni vengono interessati il **gestore della concorrenza** e il **gestore dell'affidabilità**.
Le **transazioni** rappresentano le **unità di lavoro elementare** (le insiemi di istruzioni SQL) che **modificano il contenuto di un DB**. Sintatticamente, una transazione è contornata dai comandi `BEGIN TRANSACTION` e `END TRANSACTION`, e all'interno possono comparire i comandi di `COMMIT WORK` e `ROLLBACK WORK`.

```
BEGIN TRANSACTION
UPDATE SalariImpiegati
SET Conto = Conto - 10
WHERE (CodiceImpiegato = 123)
IF (Conto > 10) COMMIT WORK
ELSE ROLLBACK WORK
```

ACID Le transazioni devono godere delle proprietà ACID:

Atomicità: la transazione deve essere eseguita con la regola **tutto o niente**.

Consistenza: la transazione deve **lasciare il DB in uno stato consistente**, quindi **eventuali vincoli d'integrità non devono essere violati**.

Isolamento: ogni transazione deve essere **eseguita in maniera indipendente dalle altre**

Persistenza (**Durability**): l'**effetto di una transazione che ha fatto** `COMMIT WORK` **non deve essere perso**.

Funzioni del DBMS in breve

Gestione dei dati: cura la memorizzazione permanente dei dati ed il loro accesso

Gestione del buffer: cura il trasferimento dei dati da memoria di massa a memoria centrale, e il caching dei dati in memoria centrale

Ottimizzazione delle interrogazioni: seleziona il piano di accesso di costo ottimo con cui valutare ciascuna interrogazione

11.4 Gestione dell'affidabilità

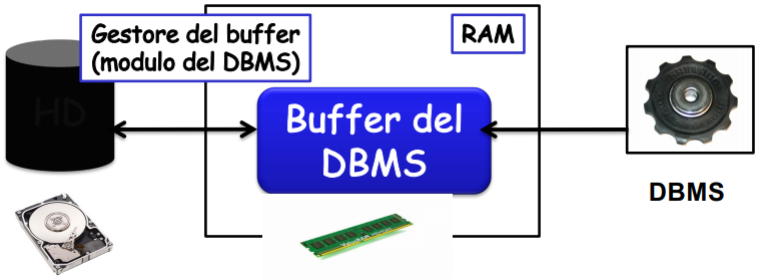
Una transazione è un'unità logica di elaborazione che corrisponde ad una serie di operazioni fisiche elementari (letture e scrittura) sul DB. Una **funzionalità essenziale** di un DBMS è la **protezione dei dati da malfuzionamenti e interferenze** dovute all'accesso contemporaneo ai dati da parte di più utenti.
Per il **programmatore**, una **transazione** è un **programma sequenziale che il sistema deve eseguire garantendo**:

Atomicità: le transazioni che terminano prematuramente (**aborted transactions**) sono **trattate dal sistema come se non fossero mai iniziate**, pertanto eventuali effetti sul DB sono annullati.

Persistenza: le **modifiche** sulla base di dati di una transazione terminata normalmente sono permanenti, cioè **non sono alterabili** da eventuali malfunzionamenti.

Serializzabilità: nel caso di **esecuzioni concorrenti** di più transazioni, l'**effetto complessivo è quello di un'esecuzione seriale**.

Per aumentare l'efficiente prestazionale, tutti i DBMS utilizzano un buffer temporaneo di informazioni in memoria principale, il quale viene periodicamente scritto su memoria secondaria.



Transazione per il DBMS Una transazione può eseguire molte operazioni sui dati recuperati da un DB, ma al DBMS interessano solo quelle di lettura o scrittura del DB indicate con $r_i[x]$ e $w_i[x]$.
Un dato letto o scritto può essere un record, un campo di un record o una pagina. Per semplicità, supponiamo che sia una pagina.

Un'operazione di lettura $r_i[x]$ comporta la lettura di una pagina nel buffer, se non già presente.
Un'operazione di scrittura $w_i[x]$ comporta l'**eventuale lettura nel buffer di una pagina e la sua modifica nel buffer, ma non necessariamente la sua scrittura in memoria permanente**. Per questa ragione, in caso di malfunzionamento, si potrebbe perdere l'effetto dell'operazione.

Malfunzionamenti

Fallimenti di transazioni: non comportano la perdita di dati in memoria temporanea né persistente (es: violazione di vincoli, violazione di protezione, stallo...)

Fallimenti di sistema: comportano la perdita di dati in memoria temporanea ma non in memoria persistente (es: comportamento anomalo del sistema, blackout, guasti hardware...)

Disastri: comportano la perdita di dati in memoria permanente (es: danneggiamento di una periferica)

Il **gestore dell'affidabilità verifica che siano garantite le proprietà di atomicità e persistenza delle transazioni**. Responsabile di:

implementare i comandi di `BEGIN TRANSACTION`, `COMMIT` e `ROLLBACK`

ripristinare il sistema dopo malfunzionamenti software (**ripresa a caldo**)

ripristinare il sistema dopo malfunzionamento hardware (**ripresa a freddo**)

Primitive undo e redo Partiamo da convenzioni notazionali: data una transazione T, indicheremo con B(T), C(T) e A(T) rispettivamente i record di begin, commit e abort relativi a T, e con U(T, O, BS, AS), I(T, O, AS) e D(T, O, BS) rispettivamente i record di update, insert e delete su un oggetto O dove BS è before state e AS è after state.
I record del log associati ad una transazione **consentono di disfare e rifare le corrispondenti azioni** sul DB:

Primitive di undo: per **disfare** un'azione su un oggetto O è sufficiente ricopiare il O il valore BS (l'insert viene disfatto cancellando l'O inserito)

Primitive di redo: per **rifare** un'azione su un oggetto O è sufficiente ricopiare in O il valore AS (il delete viene rifatto cancellando l'O eliminato)

File di log Il controllore dell'affidabilità usa **un log, dove sono indicate tutte le operazioni svolte dal DBMS**. Il file si presenta come una sequenza di record di due tipi:

Record di transizione: tengono traccia delle operazioni svolte da ciascuna transizione sul DBMS. Per ogni transazione, un record di begin (B), insert (I), delete (D) e update (U) e un record di commit (C) o di abort(A)

Record di sistema: tengono traccia delle operazioni di sistema (dump, checkpoint).
Dump produce una **copia completa** del DB, effettuata in mutua esclusione con tutte le altre transazioni quando il sistema non è operativo. La copia viene memorizzata in memoria stabile (backup). Al termine del dump, viene scritto nel log un record di dump che segnala l'avvenuta esecuzione dell'operazione in un dato istante. Il sistema quindi riprende il funzionamento normale.
Checkpoint, punto di allineamento: si scrive la marca CKP sul log per indicare che tutte le operazioni che la precedono sono state effettivamente effettuate sul DB. Un modo semplice:

Si sospende l'attivazione di nuove transazioni

si completano le precedente e si allinea il DB (ovvero si scrivono su disco le pagine "sporche" del buffer)

si scrive la marca CKP nel log

si riprende l'esecuzione

Per non bloccare le transazioni, si può scrivere un BeginCKP e un EndCKP: tra di loro si copia su disco in parallelo alle normali operazioni, e le transazioni loggate nel mezzo possono essere o non essere state copiate su disco.

Contenuto:

`(T, begin)`

Per ogni operazione di modifica:

la transazione responsabile

il tipo di ogni operazione eseguita

la nuova e vecchia versione del dato modificato
`(T, write, address, oldV, newV)`

`(T, commit)` o `(T, abort)`

Regole di scrittura

Write Ahead Log (WAL): la parte BS di ogni record viene **scritta prima che la corrispondente operazione venga effettuata nella base di dati**.

Commit Precedence: la parte AS di ogni record di log viene **scritta prima di effettuare il commit della transazione**.

Ripresa

Fallimenti di transazioni: si scrive `(T, abort)` su log e si fa l'undo

Fallimenti di sistema: DB ripristinato con il comando di `Restart` (ripartenza di emergenza), a partire dallo stato al punto di allineamento, procedendo così: le **T non terminate vengono disfatte**, e le **T terminate devono essere rifatte**.

Disastri: si riporta in linea la copia più recente del DB e la si aggiorna rifacendo le modifiche delle T terminate normalmente (ripartenza a freddo)

Per la **ripresa a caldo** ci sono quattro fasi

1. Trovare l'ultimo checkpoint (percorrendo log a ritroso)
2. Costruire gli insiemi UNDO e REDO delle transazioni da disfare e rifare
3. Ripercorrere il log all'indietro, fino alla più vecchia delle transazioni in UNDO e REDO, disfacendo tutte le azioni delle transazioni in UNDO
4. Ripercorrere il log in avanti, rifacendo tutte le azioni delle transazioni REDO

Nella **ripresa a freddo** si risponde ad un guasto che provoca il deterioramento del DB:

Si ripristinano i dati dal backup

Si eseguono le operazioni registrate sul giornale fino all'istante del guasto

Si esegue una ripresa a caldo.

Ricapitolando Gli algoritmi si differenziano a seconda del modo in cui si trattano le scritture sul DB e la terminazione delle transazioni: disfare-rifare, disfare-non rifare, non disfare-rifare, non disfare-non rifare.
Per ipotesi, le scritture sul log sono riportate subito nella memoria permanente.

11.5 Gestione della concorrenza

Serializzazione Uno schedule S si dice **seriale se le azioni di ciascuna transazione appaiono in sequenza** senza essere inframezzate da azioni di altre transazioni.
$S = \{T_1, \ldots, T_n\}$
Lo **schedule seriale è ottenibile se**:

Le **transazioni sono eseguite una alla volta** (scenario irrealistico)

Le **transazioni sono completamente indipendenti** l'una dall'altra (improbabile)

ACID Proprietà di isolamento delle transazioni.
Il DBMS transazionale gestisce questi problemi garantendo la **proprietà di isolamento**: questa proprietà **garantisce che la transazione sia eseguita come se non ci fosse concorrenza**. Viene **assicurata facendo in modo che ciascun insieme di transazioni concorrenti sottoposte sia serializzabile**.

Problematica In un sistema reale, le **transazioni vengono eseguite in concorrenza** per questioni di efficienza e scalabilità. Tuttavia, l'esecuzione concorrente determina un **insieme di problematiche** che devono essere gestite. Un esempio:

```
T1 = Read(x); x = x + 1; Write(x); COMMIT WORK
```

```
T2 = Read(x); x = x + 1; Write(x); COMMIT WORK
```

Se `x` = 3, allora al termine delle due transazioni `x` = 5 (**esecuzione sequenziale**).
Nel caso di esecuzione concorrente invece si possono verificare tanti casi (ipotizzando `x` = 3 inizialmente):

Perdita di aggiornamento

T1	T2
`Read(x)`	
`x = x + 1`	
	`Read(x)`
	`x = x + 1`
	`Write(x)`
	`COMMIT WORK`
	Scrive 4
`Write(x)`	
`COMMIT WORK`	
Scrive 4	

Lettura sporca/impropria

```
T1 = Read(x); x = x + 1; Write(x); ROLLBACK WORK
T2 = Read(x); COMMIT WORK
```

T1	T2
`Read(x)`	
`x = x + 1`	
`Write(x)`	
	`Read(x)`
	`COMMIT WORK`
	Legge 4
`ROLLBACK WORK`	

Letture inconsistenti/non riproducibili

```
T1 = Read(x); Read(x); COMMIT WORK
T2 = Read(x); x = x + 1; Write(x); COMMIT WORK
```

T1	T2
`Read(x)`	
Legge 3	
	`Read(x)`
	`x = x + 1`
	`Write(x)`
	`COMMIT WORK`
`Read(x)`	
`COMMIT WORK`	
Legge 4	

Quindi l'**esecuzione concorrente di transazioni è essenziale per un buon funzionamento del DBMS**: deve garantire che l'esecuzione concorrente di transazioni avvenga senza interferenze in caso di accessi agli stessi dati.

Serialità Un'esecuzione di un insieme di transazioni $\{T_1, \ldots, T_n\}$ si dice **seriale** se $\forall$ coppia di transazioni T_i, T_j, tutte le operazioni di T_i vengono eseguite prime di qualsiasi operazione di T_j o viceversa.

Serializzabilità Un'esecuzione di un insieme di transazioni si dice **serializzabile** se produce lo stesso effetto sulla base di dati di quello ottenibile eseguendo serialmente, in un qualche ordine, le sole transazioni terminate normalmente.

Controllo della concorrenza Nella pratica i DBMS implementano tecniche di controllo di concorrenza che garantiscono direttamente la serializzabilità delle transazioni concorrenti. Tali tecniche si dividono in due classi principali:

Protocolli ottimistici: **permettono l'esecuzione sovrapposta e non sincronizzata di transazioni** ed effettuano un **controllo sui possibili conflitti generati sono dopo il commit**.
Ogni transazione effettua liberamente le proprie operazioni sugli oggetti del DB secondo l'ordine temporale con cui le operazioni sono generate.
Al **commit**, viene **effettuato un controllo** per stabilire se sono stati riscontrati eventuali conflitti e, in caso, viene **effettuato il rollback delle azioni** delle transazioni e la relativa riesecuzione. In generale, un protocollo ottimistico è **basato su 3 fasi**:

Fase di **lettura**: ogni transazione legge i valori degli oggetti del DB su cui deve operare e li memorizza in variabili locali (copie), dove sono effettuati gli eventuali aggiornamenti.

Fase di **validazione**: vengono effettuati dei controlli sulla serializzabilità nel caso che gli aggiornamenti locali delle transazioni dovessero essere propagati sul DB

Fase di **scrittura**: gli aggiornamenti delle transazioni che hanno superato la fase di validazione sono propagati definitivamente sugli oggetti del DB.

Protocolli pessimistici/conservativi: **tendono a ritardare l'esecuzione di transazioni che potrebbero generare conflitti**, e quindi anomalie, rispetto alla transazioni concorrente. Cercano di **prevenire**, e sono quelli usati nella pratica. Si dividono in due classi principali:

Metodi **basati su lock**
I DBMS commerciali usano il meccanismo dei **lock**: per poter effettuare una qualsiasi operazione di lettura/scrittura su una risorsa (tabella o valore di una cella) è **necessario aver precedentemente acquisito il controllo, cioè il lock, sulla risorsa stessa**.

Lock in lettura, per l'**accesso esclusivo**

Lock in scrittura, per la **mutua esclusione**

Ci possono essere lock a livello di riga, tabella o pagina (**multi granularità**), oltre che in lettura o scrittua (**multi modalità**).
Quando una risorsa è bloccata, le transazioni che ne richiedono l'accesso vengono messe in coda d'attesa: è un meccanismo efficace ma incide sulle prestazioni. Un esempio di richiesta e rilascio del lock:

Codice utente

```
Transazione T0:
        R(x)
        W(y)
```

Codice con lock

```
Transazione T0:
        lockR(x)
        R(x)
        unlockR(x)
        lockW(y)
        W(y)
        unlockW(y)
```

Il gestore della concorrenza del DBMS ha quindi il compito di stabilire l'ordine con cui vengono eseguite le singole operazioni per rendere serializzabile l'esecuzione di un insieme di transazioni.
Definizione: il protocollo di blocco a due fasi ristretto (**Strict Two Phase Locking**, 2PL) è definito dalle seguenti regole:

Ogni transazione prima di effettuare un'operazione acquisisce il lock corrispondente.

Transazioni diverse non ottengono lock in conflitto

I lock si rilasciano al commit → **problema**: deadlock. Esempio:

```
T1 = R(x); W(y); COMMIT WORK
T2 = R(y); W(x); COMMIT WORK
```

T1	T2
`lockR(x)`	
	`lockR(y)`
`R(x)`	
	`R(y)`
`lockW(y)`	
	`lockW(y)`
Stallo	*Stallo*

Si può gestire con **tre tecniche**:

1. Uso dei **timeout**: **ogni operazione di una transazione ha un timeout** entro il quale deve essere completata, pena l'**abort dell'intera transazione**.
   `T1 = lockR(x, 4000); R(x); lockW(y, 2000); W(y); COMMIT WORK; unlock(x); unlock(y);`
2. **Deadlock avoidance**: prevenire le configurazioni che potrebbero portare a deadlock: **lock/unlock di tutte le risorse contemporaneamente** oppure **utilizzo di timestamp/classi di priorità tra transazioni** (ma può portare a starvation)
3. **Deadlock detection**: usare **algoritmi per identificare i deadlock** e prevedere meccanismi di recovery. Ad esempio, il grafo delle richieste/risorse e abort delle transazioni coinvolte nel cicli.

Metodi **basati su timestamp**
Ad ogni transazione si associa un **timestamp** che rappresenta il momento di inizio della transazione. Ogni transazione **non può leggere o scrivere un dato scritto da una transazione con timestamp maggiore** né **può scrivere su un dato già letto da una transazione con timestamp maggiore**.

Livelli di isolamento/consistenza per ogni transazione

SERIALIZABLE assicura che:

La transazione T legge solo i cambiamenti fatti da transazioni che hanno fatto commit

Nessun valore letto o scritto da T verrà cambiato da altre transazioni finché T non è conclusa

Se T legge un insieme di valori acceduti secondo qualche condizione di ricerca, l'insieme non viene modificato da altre transazioni finché T non è conclusa

REPEATABLE READ assicura che:

La transazione T legge solo i cambiamenti fatti da transazioni che hanno fatto commit

Nessun valore letto o scritto da T verrà cambiato da altre transazioni finché T non è conclusa

READ COMMITED assicura che:

La transazione T legge solo i cambiamenti fatti da transazioni che hanno fatto commit

T non vede nessun cambiamento eventualmente effettuato da transazioni concorrenti non concluse tra i valori letti all'inizio di T

READ UNCOMMITTED
A questo livello di isolamento una transazione T può leggere modifiche fatte ad un oggetto da una transazione in esecuzione. Ovviamente l'oggetto può essere cambiato mentre T è in esecuzione: per questo **T è soggetta a eventi fantasma**.