Basi di Dati

Angelo Passarelli

December 4, 2023

Appunti basati sulle lezioni e dispense della professoressa Giovanna Rosone [1]

[1] https://pages.di.unipi.it/rosone/index.html

Contenuti

0 Introduzione

Definizione (Base di Dati)**.** Una base di dati è un insieme organizzato di dati utilizzati per il supporto allo svolgimento di attività.

Struttura dei Dati I dati sono organizzati in insiemi strutturati che possono presentare fra loro delle relazioni. Tuple che rappresentano dati nello stesso insieme devono essere omogenee ed univoche.

Definizione (Sistema Informativo)**.** Un sistema informativo è una combinazione di risorse umane e/o materiali e procedure organizzate per la raccolta, l'archiviazione, l'elaborazione e lo scambio di informazioni necessarie ad un'attività, le quali possono essere classificate in:

- Informazioni di servizio (operative).
- Informazioni di controllo (pianificazione e gestione).
- Informazioni di governo (pianificazione strategica).

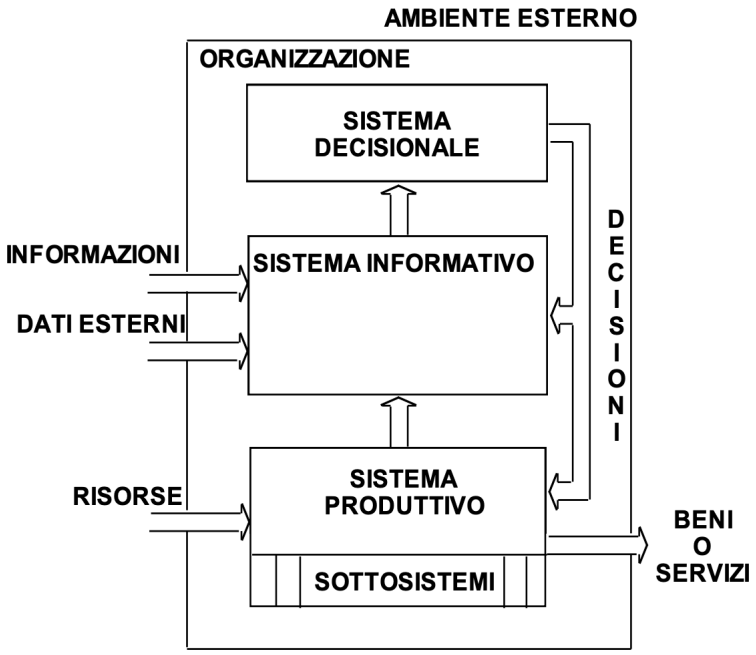


Figura 1: Esempio di sistema informativo

Definizione (Sistema Informativo Automatizzato)**.** Un sistema informativo automatizzato è una parte del sistema informativo che permette di implementare le procedure che si occupano della gestione delle informazioni usando un sistema informatico.

Definizione (Sistema Informatico)**.** Un sistema informatico è l'insieme delle tecnologie a supporto per le attività di un'organizzazione. Si possono classificare in:

- **Sistemi informatici operativi**: questi sistemi si utilizzano per svolgere le normali attività dell'azienda per la fruizione del suo bene o servizio, e per la gestione interna dei singoli reparti dell'azienda. Le operazioni sui dati in questo sistema sono di tipo OLTP (*On-Line Transaction Processing*) e prevedono elaborazioni semplici che coinvolgono pochi dati che vengono aggiornati molto frequentemente.
- **Sistemi informatici direzionali**: i dati sono organizzati in *Data Warehouse* che consentono di aiutare l'azienda nei processi di controllo delle prestazioni e di decisione manageriale. Le elaborazioni su questo tipo di sistema si chiamano OLAP (*On-Line Analytical Processing*) e prevedono l'utilizzo di una grande mole di dati che sono per lo più storici. In questo caso i dati vengono aggiornati molto raramente, ma su di essi vengono svolte molte operazioni, anche da un punto di vista multidimensionale, ovvero vengono incrociati più dati per analizzare le informazioni ottenute sotto molteplici punti di vista.

Definizione (DBMS)**.** Un *Database Management System* è un sistema che garantisce il controllo e la gestione di dati per renderli accessibili agli utenti opportuni in base ai loro privilegi. Il DBMS fornisce anche dei linguaggi che permettono di definire lo schema di un database, di scegliere le strutture dati opportune per la memorizzazione dei dati, di rispettare i vincoli per ogni tipo di dato e di poter modificare e interrogare il database.

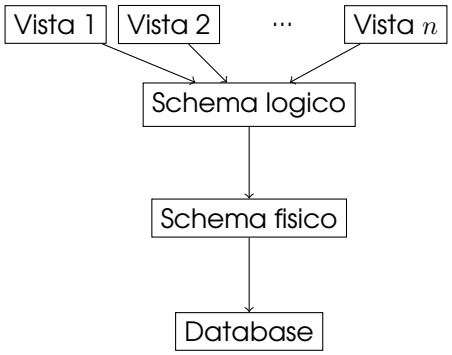
Metadati All'interno del database sono anche memorizzati dei metadati che si riferiscono agli utenti e allo schema utilizzato dal database stesso. Anche i metadati possono essere interrogati e modificati.

1 DBMS e Linguaggi

Si distinguono tre diversi livelli di descrizione dei dati:

- A livello di *vista* logica: descrive come deve apparire il database a seconda dell'utente che lo usa, in base ai suoi permessi.

- A livello logico: descrive la struttura degli insiemi dei dati e le relazioni fra essi, senza doversi occupare della loro organizzazione nella memoria.
- A livello fisico: viene descritto come sono organizzati fisicamente i dati nella memoria e vengono riportate quali strutture dati ausiliare vengono utilizzate.



Quest approccio permette di garantire le proprietà di indipendenza logica e fisica:

- **Indipendenza Logica**: gli applicativi non necessitano modifiche in seguite a variazioni dello schema logico.
- **Indipendenza Fisica**: gli applicativi non necessitano modifiche in seguito a cambiamenti dell'organizzazione fisica dei dati.

Per quanto riguarda i linguaggi di interrogazione, questi possono essere distinti in:

- **DML** (Data Manipulation Language): per l'interrogazione e l'aggiornamento dei dati.
- **DDL** (Data Definition Language): per la definizione di schemi, sia logici che fisici, ed altre operazioni.

1.1 Funzionalità del DBMS

Un DBMS deve prevedere più modalità d'uso per soddisfare le esigenze di più categorie di utenti che accedono al database. Deve poter offrire:

- Un'interfaccia grafica per accedere ai dati.
- Un linguaggio di interrogazione per gli utenti inesperti (non programmatori).

- Un linguaggio di programmazione per chi sviluppa applicazioni, nello specifico deve prevedere l'integrazione del *DDL* e del *DML* nel linguaggio ospite.
- Un linguaggio per lo sviluppo di interfacce per le applicazioni.
- Predisporre per l'amministratore strumenti per stabilire i diritti d'accesso ai dati, per il ripristino del sistema e per la modifica e la definizione degli schemi logici (sia interno che esterno).

1.2 Proprietà dei Database

Il DBMS permette di garantire al database le seguenti proprietà:

- Integrità: mantenimento dei vincoli d'integrità dichiarati in fase di definizione dello schema.
- Affidabilità: protezione dei dati da parte di malfunzionamenti sia software che hardware e da anomalie indesiderate come l'accesso concorrente al database da parte di più utenti.
- Sicurezza: protezione dei dati da parte di utenti non autorizzati.

Inoltre un DBMS deve essere in grado di gestire collezioni di dati che siano:

- Grandi
- Persistenti: il periodo di vita dei dati è indipendente dai programmi che li utilizzano.
- Condivise: possono essere usati da programmi diversi.

Il DMBS deve essere anche efficiente (utilizzando al meglio le risorse in termini di *spazio* e *tempo*) ed efficace.

1.3 Transazioni

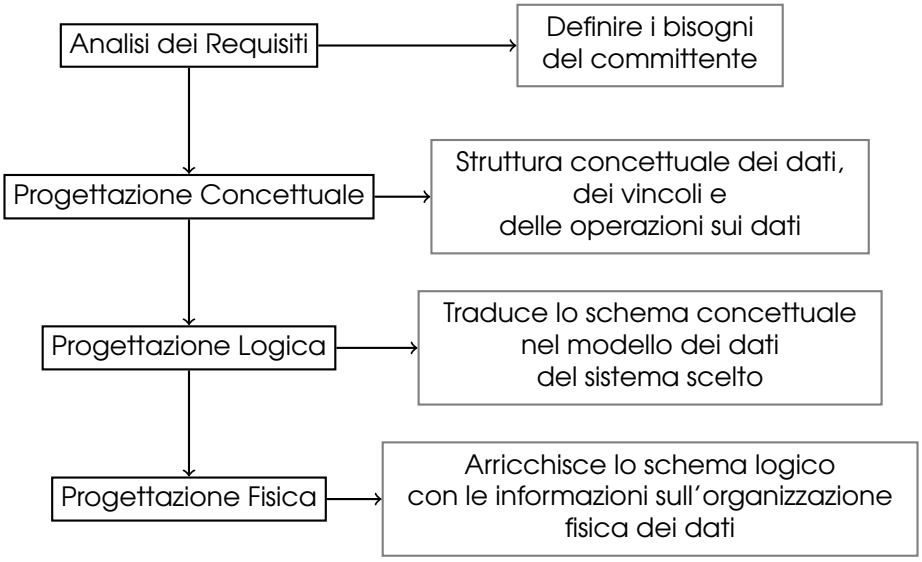
Definizione (Transazione)**.** Una transazione è una serie di azioni di lettura e scrittura sulla memoria permanente o di elaborazione dati in memoria temporanea. Presenta le seguenti proprietà:

- **Atomicità**: le transizioni che non vanno a buon fine o che vengono abortite sono trattate come se non fossero mai state eseguite.
- **Persistenza**: le modifiche effettuate da una transizione andata a buon fine sono permanenti, ovvero non possono essere alterate da malfunzionamenti.
- **Serializzabilità**: nel caso di esecuzioni concorrenti di più transazioni, l'effetto ottenuto è quello di un'esecuzione seriale.

2 Modellazione

Definizione (Modello Astratto)**.** Un modello astratto è la rappresentazione formale di idee e conoscenze relative ad un fenomeno.

La modellazione è centrale nella progettazione del database che comprende le fasi presenti in figura.



2.1 Fasi della Modellazione

2.2 Analizzare il Dominio

Il dominio può presentare più aspetti da dover analizzare:

- Aspetto *ontologico*: conoscere ciò che si suppone esista nell'universo del contesto e quindi ciò che è da modellare. Occorre analizzare tre tipi di conoscenze:
  - Conoscenza *concreta*: le entità del contesto e le associazioni fra di esse.

    Definizione (Entità)**.** Sono oggetti di cui occorre definire le proprietà.

    Definizione (Proprietà)**.** Descrivono le caratteristiche di determinate entità e sono formata da una coppia `(Attributo, Valore)`. Ogni proprietà ha ad essa associato un dominio, quindi un insieme di valori che può assumere. Inoltre le proprietà si possono classificare in:

    - Atomiche o Strutturate: atomiche se il loro valore non è ulteriormente scomponibile.
    - Totali o Parziali: se è obbligatoria oppure opzionale.
    - Univoche o Multivalore: univoca se per ogni entità la scelta del valore è unico (*es. codice fiscale*).
    - Costanti o Variabili
    - Calcolate o Non Calcolate: calcolata se è possibile derivarla da altre proprietà.

    Definizione (Tipo di un'entità)**.** Ogni entità appartiene ad un tipo che ne indica la propria natura.

    Definizione (Collezione)**.** Insieme di entità dello stesso tipo.

  - Conoscenza *astratta*: la struttura e i vincoli sulle entità.
  - Conoscenza *procedurale*: le operazioni di base, sia dei singoli utenti e sia come avviene la comunicazione con il sistema informatico.

- Aspetto *logico*: meccanismi di astrazione (*modello di dati, per es. diagrammi E-R* [2]) con cui descrivere la struttura della conoscenza concreta.
- Aspetto *linguistico*: linguaggio formale con cui definire il modello.
- Aspetto *pragmatico*: insieme di regole da seguire in fase di modellazione.

2.3 Oggetti e Classi

Definizione (Oggetto)**.** Un oggetto è un'entità software che presenta uno *stato*, un *comportamento* e un'*identità*. Lo stato è rappresentato da un insieme di costanti o variabili, mentre il comportamento è un insieme di procedure locali chiamate *metodi*. Un oggetto può rispondere a dei messaggi di input, con dei valori memorizzati nello stato o calcolandoli con un metodo.

Definizione (Classe)**.** Una classe è un insieme di oggetti dello stesso tipo, e presenta delle operazioni per l'inserimento e la rimozione di elementi.

Definizione (Tipo Oggetto)**.** Un tipo oggetto definisce l'insieme degli attributi a cui può combaciare un insieme di possibili oggetti. I tipi oggetto non sono presenti nei diagrammi E-R, però dagli attributi di una collezione è possibile dedurre il tipo oggetto associato.

[2] `https://it.wikipedia.org/wiki/Modello_E-R`

2.4 Associazioni

Definizione (Istanza di un'associazione)**.** Un'istanza di un'associazione determina un legame logico tra due o più istanze.

Definizione (Associazione)**.** Un'associazione $R(X, Y)$ fra due collezioni di entità chiamate X e Y è un insieme, che varia nel tempo, di istanze di associazione tra gli elementi delle due collezioni. Il prodotto cartesiano $(X \cdot Y)$ è chiamato dominio dell'associazione.

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

Definizione (Vincolo di Unicità)**.** Un'associazione $R(X, Y)$ è detta univoca rispetto ad X se per ogni elemento di $x \in X$ esiste al più un elemento $y \in Y$ che è associato ad x. Se questo vincolo non vale, si dice che l'associazione è multivalore rispetto ad X.

Cardinalità dell'Associazione

- $R(X, Y)$ è **(1:N)** se è multivalore sy X ed univoca su Y.
- $R(X, Y)$ è **(N:1)** se è univoca sy X e multivalore su Y.
- $R(X, Y)$ è **(N:M)** se è multivalore sy X e multivalore su Y.
- $R(X, Y)$ è **(1:1)** se è univoca sy X ed univoca su Y.

Definizione (Vincolo di Totalità)**.** Un'associazione $R(X, Y)$ è detta totale su X se per ogni elemento $x \in X$ esiste almeno un elemento $y \in Y$ associato ad x. Se questo vincolo non vale, si dice che l'associazione è parziale rispetto ad X.

Rappresentazione delle Associazioni Un'associazione fra due collezioni C_1 e C_2 si rappresenta con una linea che collega le due classi. La linea si etichetta con il nome dell'associazione. L'univocità di una classe C_1 si rappresenta disegnando una freccia singola sulla linea che esce và da C_1 a C_2. Se invece l'associazione è multivalore si indica con una doppia freccia. La parzialità invece è rappresentata con un taglio sulla linea vicino alla freccia, mentre la totalità è rappresentata dall'assenza del taglio.

Figura 2: In questo caso abbiamo un'associazione *multivalore* da entrambe la parti, ma *parziale* per C_2 e *totale* per C_1

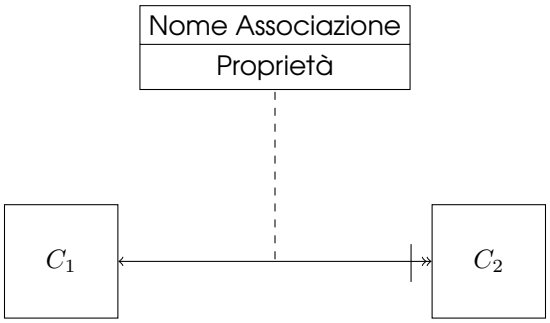


Figura 3: Le associazioni possono presentare proprietà

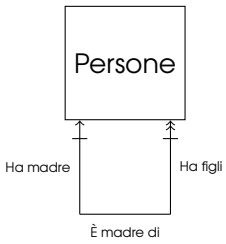


Figura 4: O possono anche essere ricorsive. In questo caso occorre etichettare l'associazione non solo con il proprio nome, ma anche con i nomi dei ruoli che hanno le due entità nell'associazione.

Associazione Non Binarie Le associazioni non binarie, per semplicità, non vengono rappresentate graficamente, ma ad esempio, per quanto

riguarda quelle *ternarie*, queste vengono trasformate in tre associazioni *binarie* aggiungendo un'altra collezione al posto dell'associazione *ternaria*.

2.5 Gerarchie

Le classi di entità possono essere organizzate in una gerarchia di *specializzazione*. Una classe della gerarchia minore viene chiamata **sottoclasse**, mentre le altre si chiamano **superclassi**. Gli elementi di una sottoclasse sono un sottoinsieme degli elementi della superclasse.

2.5.1 Tipi Oggetto

Fra i *tipi oggetto* viene definita una relazione di sottotipo, che comprende le seguenti proprietà:

- È una relazione asimmetrica, riflessiva e transitiva.
- Inoltre, se un tipo T è *sottotipo* di T', allora tutti gli elementi di T possono essere usati in tutti i contesti in cui appaiono elementi di tipo T'. Questa proprietà è chiamata sostitutività ed è data dal fatto che gli elementi di T hanno tutte le proprietà degli elementi di T', e per ogni proprietà di T', il suo tipo in T è un sottotipo di quello che ha in T'.

Ereditarietà L'*ereditarietà* è una proprietà delle gerarchie che permette di definire un *tipo oggetto* a partire da un altro. In quanto, nel nostro contesto, a partire da un tipo, si vuole solo definire un sottotipo, si parla di *ereditarietà stretta*, che permette solo:

- L'aggiunta di altri *attributi*.
- La ridefinizione di attributi del *supertipo*, però solo specializzando ulteriormente il tipo dell'attributo.

2.5.2 Classi

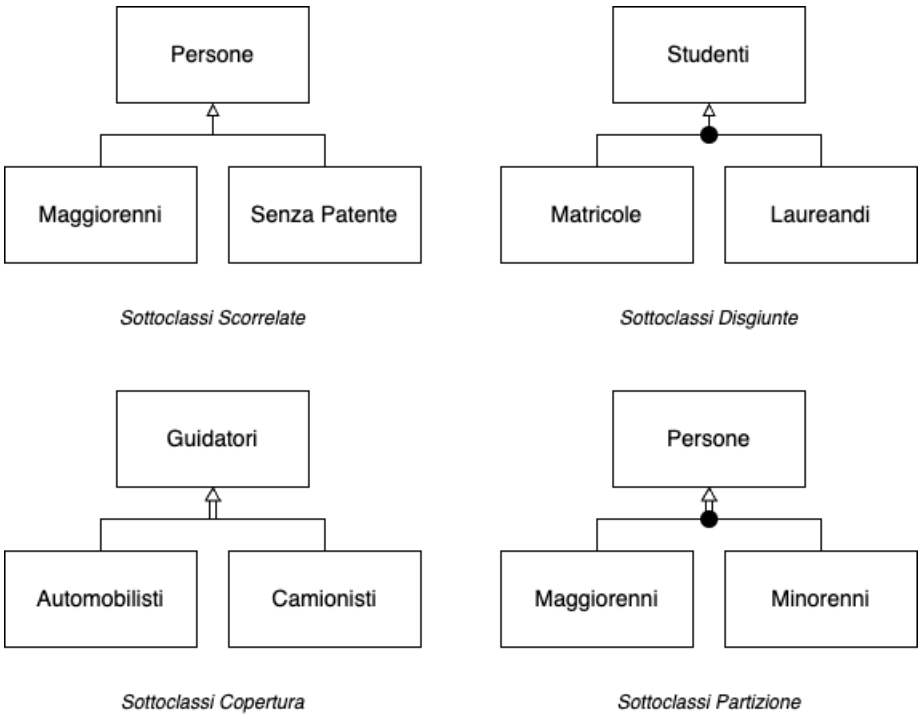
Fra le *classi*, invece, viene definita una relazione di sottoclasse, con le seguenti proprietà:

- Sempre asimmetrica, riflessiva e transitiva, come per i *tipi oggetto*.
- Se una classe C è *sottoclasse* di C', allora il tipo degli elementi di C è sottotipo del tipo degli elementi di C' (**vincolo intensionale**).
- Se C è *sottoclasse* di C', allora gli elementi di C sono un sottoinsieme degli elementi di C' (**vincolo estensionale**).

Vincoli Possiamo distinguere due tipi di vincoli su insiemi di sottoclassi:

- Disgiunzione: ogni coppia di sottoclassi nell'insieme è *disgiunta*, quindi è priva di elementi comuni.
- Copertura: l'unione degli elementi delle sottoclassi coincide con l'insieme degli elementi della *superclasse*.

Se possiedono entrambi i vincoli, allora l'insieme delle *sottoclassi* forma una partizione della *superclasse*. Altrimenti se nessuno dei due vincoli è rispettato, si dice che le sottoclassi sono scorrelate.



2.5.3 Ereditarietà Multipla

Un tipo può anche essere definito per *ereditarietà* anche a partire da più di un *supertipo*. Questo però può creare alcuni problemi quando lo stesso *attributo* viene ereditato da più di un supertipo, ma i tipi degli attributi fra loro sono diversi.

3 Progettazione Logica

L'obbiettivo della **progettazione logica** è quello di ridefinire lo *schema concettuale* in uno *schema logico relazionale* che rappresenti gli stessi dati ma in una maniere più efficiente (minimizzando la ridondanza) e corretta, per esempio tenendo conto della dimensione dei dati e del tipo di operazioni che si effettueranno sul database. Anche perchè alcuni costrutti dello *schema relazionale* non sono rappresentabili concretamente.

3.1 Relazioni

Definizione (Relazione Matematica)**.** Una *relazione matematica* è un insieme di *n*-uple ordinate $(d_1, \ldots, d_n)$ tali che $d_1 \in D_1, \ldots, d_n \in D_n$.

Essendo un insieme, non c'è ordinamento fra le *n*-uple, che inoltre devono essere tutte distinte. Però l'ordinamento all'interno della *n*-upla conta, infatti l'*i*-esimo valore deve provenire dall'*i*-esimo dominio. Per questo si dice che la struttura della relazione è *posizionale*.

Definizione (Attributo)**.** A ciascun dominio della relazione si associa un nome, chiamata **attributo**.

Definizione (Tupla)**.** Una **tupla** su un insieme di *attributi* chiamato X, è una funzione t che associa a ciascun *attributo* un valore del suo *dominio*.

Una *relazione su* X è un insieme di *tuple* su X.

3.2 Tabelle

Una **tabella** rappresenta una relazione se:

- I valori di ogni *colonna* sono fra loro *omogenei*.
- Le *righe* sono tutte *diverse* fra loro.
- Le *intestazioni* delle colonne sono tutte *diverse* fra loro.

In una *tabella* l'ordinamento tra le righe e le colonne è irrilevante.
In ogni tabella è possibile distinguere due parti:

- Lo **schema** è rappresentato dalle intestazioni della tabella, che sono invarianti nel tempo e descrivono la struttura della tabella (**aspetto intensionale**).
- L'**istanza** sono i valori attuali presenti nella tabella, che possono cambiare nel tempo (**aspetto estensionale**).

Definizione (Tipo Ennupla)**.** Un tipo ennupla chiamato T è un insieme finito di coppie (*Attributo*, *Tipo Elementare*).

Definizione (Schema di Relazione)**.** Se T è un *tipo ennupla*, allora $R(T)$ è lo schema della relazione R.

Definizione (Schema di Database)**.** Lo schema di un database è un insieme di *schemi di relazioni* $R_i(T_i)$.

Definizione (Istanza di Relazione)**.** Un'istanza di relazione, anche chiamata **relazione** su uno schema $R(T)$ è l'insieme r di tuple di tipo T.

Definizione (Istanza di Database)**.** Un'istanza di database su uno schema $R = \{R_1(T_1), \ldots, R_n(T_n)\}$ è l'insieme delle *relazioni* $r = \{r_1, \ldots, r_n\}$, dove r_i è un'*istanza di relazione* su $R_i(T_i)$.

Valore Nullo Si indica con `NULL`, e indica l'assenza di un valore del dominio.

3.3 Vincoli di Integrità

Un **vincolo d'integrità** è una proprietà che deve essere soddisfatta da ogni singola *istanza* della relazione, in modo tale che rappresenti informazioni corrette per l'applicazione.

Il vincolo viene espresso mediante un predicato, che associa ad ogni *istanza* il valore *vero* o *falso*.

I vincoli si possono classificare in:

- Vincoli intrarelazionali: ovvero quelli che devono essere rispettati da valori della relazioni presa in considerazione, e possono essere:
  - Vincoli sul dominio, che coinvolgono un solo *attributo*.
  - Vincoli di ennupla, che esprimono condizioni sui valori di ogni ennupla, indipendentemente dalle altre ennuple.
- Vincoli interrelazionali: sono quei vincoli che devono essere rispettati da valori presenti in relazioni diverse.

3.4 Chiavi

Definizione (Superchiave)**.** Un insieme K di attributi viene chiamato **superchiave** per una relazione r, se r non contiene due ennuple distinte t_1, t_2 tali che $t_1[K] = t_2[K]$.

Definizione (Chiave)**.** K invece viene definito **chiave** per r se è una *superchiave minimale* per r, ovvero non deve contenere altre *superchiavi*.

Nota Bene Dato che una *relazione* non può contenere ennuple con valori uguali, allora per ogni *relazione* esiste sempre una superchiave rappresentata dall'insieme di tutti gli attributi su cui è definita.

Definizione (Chiave Primaria)**.** Una **chiave primaria** è una *chiave* sulla quale non sono ammessi valori nulli.

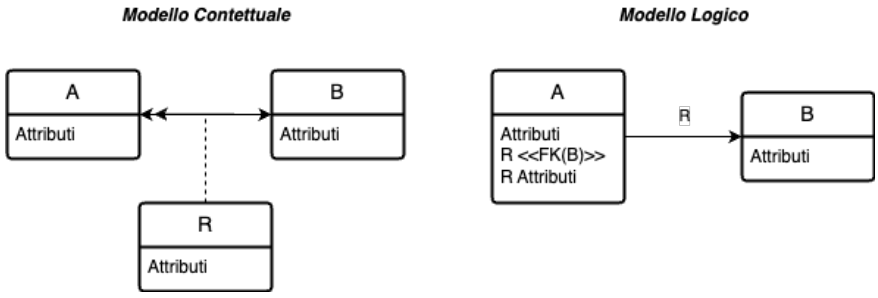
Nel modello relazionale per correlare due relazioni diversi si usano i valori delle *chiavi primarie*.

Definizione (Integrità Referenziale)**.** Un vincolo di **integrità referenziale**, anche chiamato *foreign key*, fra alcuni attributi X di una relazione R_1 e un'altra relazione R_2 impone ai valori di X di comparire come valori della *chiave primaria* di R_2.

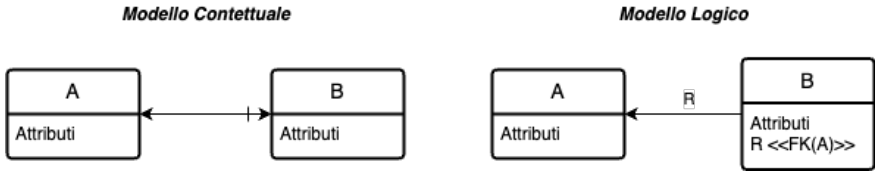
Violazioni Se per esempio si provasse ad eliminare una ennupla dalla tabella che viene riferita si verificherebbe un rifiuto dell'operazione, nel caso in cui la *chiave primaria* di quella ennupla viene riferita da altre ennuple di altre tabelle. Quindi in questo caso occorre procedere con un'*eliminazione a cascata*, ovvero si impostano prima a `NULL` i valori degli attributi delle ennuple delle tabelle che contengono riferimenti a quella *chiave primaria*; successivamente si può procedere con l'eliminazione della ennupla che adesso non verrà più riferita.

3.5 Rappresentazione delle Associazioni

Uno a Molti Le associazioni *uno a molti* si rappresentano aggiungendo, agli attributi della relazione rispetto alla quale l'associazione è univoca, una *chiave esterna* che si riferisce all'altra relazione. Nel caso in cui l'associazione ha degli attributi si aggiungono anch'essi alla relazione.

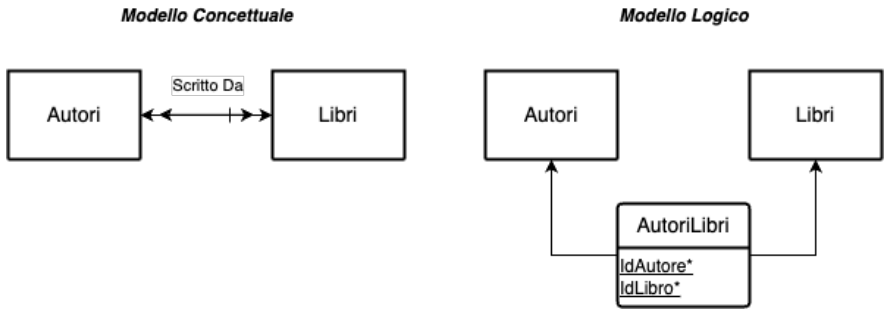


Uno a Uno In questo caso si aggiunge la *chiave esterna* scegliendo arbitrariamente una delle due relazioni ma, in caso in cui esiste un vincolo di totalità, si preferisce la relazione rispetto alla quale l'associazione è *totale*.

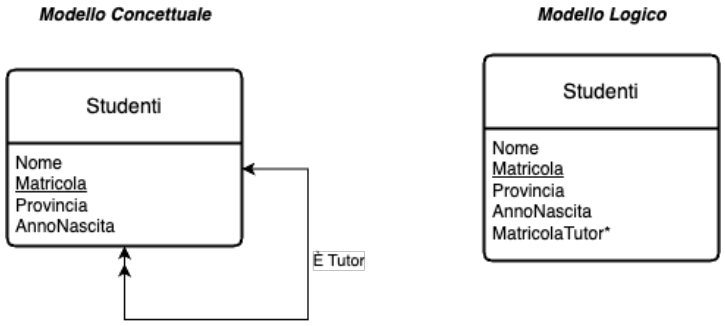


Vincoli sulle Cardinalità La direzione dell'associazione rappresentata dalla *chiave esterna* è chiamata la *diretta* dell'associazione. Per imporre un vincolo di univocità della diretta occorre definire un vincolo di chiave sulla *chiave esterna*; mentre per descrivere un vincolo di totalità della diretta si impone un vincolo `NOT NULL` sempre sulla *chiave esterna*.

Molti a Molti Un'associazione *molti a molti* si traduce aggiungendo tra le due relazioni, una terza che ha come attributi (e come *chiave primaria*) le *chiavi primarie* delle due relazioni.

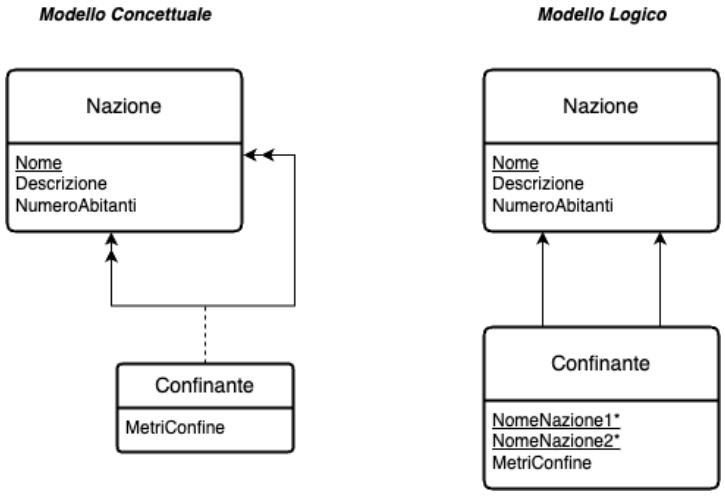


Ricorsione In questo caso la *chiave esterna* si aggiunge alla stessa e sola relazione.



Ricorsione Molti a Molti Anche qui si costruisce una seconda relazione che ha come *chiave primaria* le due *chiavi primarie* delle due istanze

coinvolte.



3.6 Rappresentazione delle Gerarchie

Le *gerarchie* non possono essere rappresentate direttamente, quindi vanno eliminate sostituendole con altre classi e relazioni.

Ci sono tre metodi:

- **Relazione Unica**: se A_0 è la classe genitore di A_1 e A_2, queste vengono eliminate e accorpate ad A_0. Ad A_0 viene aggiunto un attributo chiamato discriminatore, che indica per ogni istanza da quale classe figlia deriva. Ovviamente anche gli attributi delle classi figlie vengono assorbite dal genitore e assumono valore `NULL` sulle istanze di una classe figlia che non possedeva quegli attributi. Per quanto riguarda le relazioni, invece, anche queste vengono assorbite dalla classe genitore ma avranno comunque cardinalità minima uguale a 0, anche qui per le istanze di una classe figlia che non aveva quella relazione.

- **Partizionamento Orizzontale**: la classe genitore A_0 viene eliminata e le classi figlie A_1 e A_2 ereditano gli attributi e le relazioni del genitore. L'unico caso in cui non si può adoperare questo metodo è quando è presente un *vincolo referenziale* verso la classe genitore, in quanto non è possibile spezzare il vincolo in più relazioni diverse.

- **Partizionamento Verticale**: la gerarchia si trasforma in tante associazioni *uno a uno* che legano ogni classe figlia con la classe genitore. In questo caso vanno aggiunti dei vincoli, ovvero ogni istanza di A_0 non può partecipare a tutte le associazioni, ma ad al più una; e nel caso in cui la gerarchia è totale deve essere esattamente 1.

Nota Bene In quest'ultimo metodo la *chiave primaria* della classe genitore è sia *chiave esterna* che *chiave primaria* per le figlie.

3.7 Rappresentazione Campi Multivalore

Per la gestione dei *campi multivalore* viene creata una nuova relazione che ha come attributi il nome del campo e una *chiave esterna* che si riferisce alla relazione in cui si trovava. La *chiave primaria* di questa nuova relazione è l'intera ennupla.

4 Algebra Relazionale

Definizione (Algebra Relazionale)**.** Con **algebra relazionale** si intende un insieme di *operatori* su relazioni che danno come risultato altre relazioni. Non viene usato come linguaggio di interrogazione dei *DBMS,* ma come rappresentazione interna delle interrogazioni.

4.1 Operatori Insiemistici

Le relazioni vengono viste come degli *insiemi*. Gli operatori di *unione*, *differenza* ed *intersezione* sono applicabili solo a relazioni definite sugli stessi *attributi*.

- **Unione**: l'unione di due relazioni R e S definite sullo stesso insieme di attributi X è indicata con $R \cup S$ ed è una relazione su X che contiene le tuple che appartengono a R o a S senza ripetizioni.
- **Differenza**: la differenza è indicata con $R - S$ ed è una relazione sempre su X che contiene tutte le tuple che appartengono ad R ma non a S.
- **Ridenominazione**: è un operatore *monadico*, ovvero su una sola relazione, che permette di modificare lo schema, ridenominando il nome di uno o più attributi, ma lasciando inalterate le istanze.

$$\rho_{Nuovo\ Nome\ Attributo \leftarrow Vecchio\ Nome\ Attributo}(Relazione)$$

- **Proiezione**: è un altro operatore *monadico* che produce un risultato su un sottoinsieme degli attributi della relazione e contiene le ennuple della relazione ristrette solo agli attributi del sottoinsieme.

$$\pi_{Lista\ Attributi}(Relazione)$$

La cardinalità di una *proiezione* è basata sul sottoinsieme X degli attributi:

  - Se X è una *superchiave* di R, allora $\pi_X(R)$ contiene esattamente tante ennuple quante ne contiene R.
  - Altrimenti se X non è *superchiave*, potrebbero essere presenti dei valori che su quegli attributi sono ripetuti e che quindi vengono rappresentati una sola volta.

  Nella *proiezione* è possibile ridenominare un attributo nella $Lista\ Attributi$ scrivendo `Nome Vecchio as Nome Nuovo`.

- **Selezione** o **Restrizione**: è sempre un operatore *monadico* che produce una relazione con lo stesso schema dell'operando, che contiene un sottoinsieme delle sue ennuple che soddisfano una data condizione.

  $$\sigma_{\ Cond}\ (R)$$

  Con $Cond$ generata dalla seguente grammatica:

  ```
  Cond ::= Exp Theta Exp | Cond And Cond | Cond Or Cond | Not Cond
  Exp ::= Attributo | Costante | Exp Op Exp
  Theta ::= = | < | > | != | <= | >=
  Op ::= + | - | * | StringConcat
  ```

  Le condizioni atomiche si riferiscono solo ai valori non nulli, per riferirsi anche ai valori nulli si usano forme di condizioni come `IS NULL` e `IS NOT NULL`.

- **Prodotto**: operatore fra più relazioni che devono avere fra loro tutti i nomi degli attributi distinti, e che esegue il prodotto cartesiano fra gli insiemi di tuple.

- **Intersezione**: l'intersezione è indicata con $R \cap S$ ed è una relazione definita sullo stesso insieme di attributi X che contiene le tuple che appartengono sia ad R che ad S. L'*intersezione* è chiamato operatore *derivato*, dato che è possibile derivarlo usando altri operatori:

  $$R \cap S\ =\ \{x \mid x \in R \wedge \exists\ y \in S\ t.c.\ x = y\}$$

  allora se per esempio R ed S sono definiti con $X = \{A, B\}$.

  $$\pi_{A,B}(\sigma_{A=S.A\ AND\ B=S.B}(R \times \rho_S(S)))$$

  Partendo dall'interno, si ridenominano tutti gli attributi di S ponendogli davanti il prefisso S, successivamente viene fatto il *prodotto* con R e tramite la *selezione* selezioniamo solo le tuple che hanno gli attributi semanticamente uguali con gli stessi valori. Infine si fà una *proiezione* per eliminare gli attributi ridondanti.

4.1.1 Join

Il **join** o **giunzione** permette di correlare dati in relazioni diverse. Anch'esso è un operatore derivato in quanto si può ottenere per composizione degli operatori visti precedentemente. Si indica con il simbolo $\bowtie$.

Il **join naturale** produce un'unione degli attributi dei due operandi, combinando le ennuple degli operandi che hanno valori uguali sugli attributi in comune. Ad esempio $R_1 \bowtie R_2$ è una relazione su $X_1 \cup X_2$ definita come segue:

$$R_1 \bowtie R_2 = \{t \; su \; X_1 \cup X_2 \mid esistono \; t_1 \in R_1 \; e \; t_2 \in R_2 \\ con \; t[X_1] = t_1 \; e \; t[X_2] = t_2\}$$

Un *join* viene chiamato *completo* se ogni ennupla delle due relazioni contribuisce al risultato. Al contrario si parla di *join non completo*. Generalmente possiamo definire un limite inferiore e superiore alla cardinalità di una join:

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

Nel caso il join è *completo* possiamo restringere il limite inferiore dicendo che $|R_1 \bowtie R_2| \geq \max\{|R_1|, |R_2|\}$.

Nel caso in cui il join coinvolge come attributo comune una *chiave* di R_2, allora:

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

Se oltre a coinvolgere una *chiave* di R_2, vi è un vincolo di *integrità referenziale* tra un attributo di R_1 e la chiave di R_2 allora la cardinalità della join è esattamente $|R_1|$.

Un **join esterno** estende con valori nulli le ennuple che verrebbero tagliate fuori da un *join naturale*. Esiste in tre versioni:

- Join esterno sinistro $R \overset{\leftarrow}{\bowtie} S$: mantiene tutte le ennuple del primo operando, estendendole con valori nulli se necessario.
- Join esterno destro $R \overset{\rightarrow}{\bowtie} S$: mantiene tutte le ennuple del secondo operando, estendendole con valori nulli se necessario.
- Join esterno completo $R \overset{\leftrightarrow}{\bowtie} S$: mantiene tutte le ennuple di entrambi gli operandi, estendendole con valori nulli se necessario.

Nota Bene Il *prodotto cartesiano* si può vedere come un *join naturale* su relazioni che non hanno attributi in comune.

Dato che il *prodotto cartesiano* non ha senso se non lo facciamo seguire da una selezione (che permette di specificare la condizione sui campi che semanticamente rappresentano lo stesso attributo), si usa l'operazione di **theta-join** definito come segue:

$$R_1 \bowtie_{Condizione} R_2$$

che è equivalente a:

$$\sigma_{Condizione} (R_1 \bowtie R_2)$$

N.B.: dato che R_1 ed R_2 non hanno attributi comuni, in questo caso $\bowtie = \times$.

Se nel *theta-join* l'operatore di confronto nella $Condizione$ è sempre l'uguaglianza, allora si parla di **equi-join**.

Self Join Supponiamo di avere questa relazione `Genitori`:

Genitore	Figlio
Luca	Anna
Maria	Anna
Giorgio	Luca
Silvia	Maria
Enzo	Maria

e di voler ottenere una relazione `Nonno-Nipoti`. Per far ciò occorre riutilizzare la stessa tabella ma facendo $Genitori \bowtie Genitori$ si otterrebbe di nuovo la tabella `Genitori`. In questo caso occorre ridenominare la tabella due volte:

Nonno	Genitore
Luca	Anna
Maria	Anna
Giorgio	Luca
Silvia	Maria
Enzo	Maria

$\rho_{Nonno,\, Genitore \leftarrow Genitore,\, Figlio}(Genitore)$

Genitore	Nipote
Luca	Anna
Maria	Anna
Giorgio	Luca
Silvia	Maria
Enzo	Maria

$\rho_{Nipote \leftarrow Figlio}(Genitore)$

Infine, effettuando una *join* seguita da una *proiezione* otteniamo:

Nonno	Nipote
Giorgio	Anna
Silvia	Anna
Enzo	Anna

$$\pi_{Nonno,\, Nipote}(\rho_{Nonno,\, Genitore \leftarrow Genitore,\, Figlio}(Genitore) \bowtie \rho_{Nipote \leftarrow Figlio}(Genitore))$$

4.2 Raggruppamento

Il **raggruppamento** è definito dall'espressione ${}_{\{A_i\}}\gamma_{\{f_i\}}(R)$, dove gli A_i sono gli attributi di R mentre le f_i sono funzioni di aggregazione. Il *raggruppamento* prima partiziona le ennuple di R mettendo nello stesso gruppo le ennuple con valori uguali solo sui campi degli A_i, successivamente si calcolano le funzioni f_i per ogni partizione. Infine per ogni partizione verrà restituita una solo ennupla che avrà come attributi i valori degli A_i e i valori delle funzioni f_i.

4.3 Trasformazioni Algebriche

Permettono di scegliere diversi ordini di *join* e di anticipare *proiezioni* e *selezioni* per lavorare su tabelle più piccole.

Idempotenza delle Proiezioni

$$\pi_A(\pi_{A,B}(R)) \equiv \pi_A(R) \tag{1}$$

Atomizzazione delle Selezioni

$$\sigma_{Cond_1}(\sigma_{Cond_2}(R)) \equiv \sigma_{Cond_1 \wedge Cond_2}(R) \tag{2}$$

Atomizzazione della Selezione rispetto al Join
Pushing Selection Down

$$\sigma_{Cond}(R \bowtie S) = R \bowtie \sigma_{Cond}(S) \tag{3}$$

Se $Cond$ fà riferimento solo agli attributi di S.

Anticipazione della Proiezione rispetto al Join
Pushing Projections Down

$$\pi_{X_1 Y_2}(R \bowtie S) = R \bowtie \pi_{Y_2}(S) \tag{4}$$

Se R è definito su X_1 e S su X_2, $Y_2 \subseteq X_2$ e gli attributi in $X_2 - Y_2$ non sono coinvolti nel join.

Distributività della Selezione

$$\sigma(R \cup S) = \sigma(R) \cup \sigma(S) \tag{5}$$

$$\sigma(R - S) = \sigma(R) - \sigma(S) \tag{6}$$

Distributività della Proiezione

$$\pi_X(R \cup S) = \pi_X(R) \cup \pi_X(S) \tag{7}$$

Nota Bene La *proiezione* sulla *differenza* non gode della proprietà di *distributività*:

$$\pi_X(R - S) <> \pi_X(R) - \pi_X(S) \tag{8}$$

Inglobamento della Selezione nella Join

$$\sigma_{Cond}(R \bowtie S) \equiv R \bowtie_{Cond} S \tag{9}$$

$$\sigma_{Cond_1 \wedge Cond_2}(R \times S) \equiv \sigma_{Cond_1}(R) \times \sigma_{Cond_2}(S) \tag{10}$$

$$\sigma_{Cond_1 \vee Cond_2}(R) \equiv \sigma_{Cond_1}(R) \cup \sigma_{Cond_2}(R) \tag{11}$$

$$\sigma_{Cond_1 \wedge Cond_2}(R \times S) \equiv \sigma_{Cond_1}(R) \cap \sigma_{Cond_2}(R) \tag{12}$$

$$\sigma_{Cond_1 \wedge \neg Cond_2}(R \times S) \equiv \sigma_{Cond_1}(R) - \sigma_{Cond_2}(R) \tag{13}$$

$$R \times (S \times T) \equiv (R \times S) \times T \tag{14}$$

$$(R \times S) \equiv (S \times R) \tag{15}$$

$$\sigma_{Cond}({}_X\gamma_F(R)) \equiv {}_X\gamma_F(\sigma_{Cond}(R)) \tag{16}$$

4.4 Operatori Non Insiemistici

Proiezione MultiInsiemistica $\pi^b_{A_i}(R)$, la b sta ad indicare che le tuple duplicate non vanno eliminate.

Ordinamento $\tau_{A_i}(R)$, ordina i valori degli attributi di ogni tupla seguendo l'ordine degli A_i.

5 SQL

5.1 Query Language

Le interrogazioni al database vengono fatte mediante il comando `SELECT`, nel quale occorre specificare la lista degli attributi interessati. Il comando presenta anche due clausole, `FROM` che indica in quali tabelle sono contenuti i dati, e `WHERE` (*opzionale*) per esprimere le condizioni che i dati devono soddisfare.

```
SELECT Lista Attributi
FROM Lista Tabelle
[WHERE Condizione]
```

Specificare la *Lista Attributi*, anche chiamata **target list** rappresenta l'operazione di *proiezione*. La clausola `WHERE`, invece permette di implementare la *selezione*.

Il **theta-join**, invece è implementato indicando nella clausola `FROM` le due tabelle, e nel `WHERE` la condizione che stabilisce quali righe accoppiare.

```
SELECT *
FROM Studenti, Esami
WHERE Studenti.Matricola = Esami.Studenti
```

Nel caso in cui la clausola `WHERE` non è presente, la query sopra indicata esegue il prodotto cartesiano anche chiamato *cross-join*.

DISTINCT *SQL* è un linguaggio che lavora su *multiinsiemi*, quindi di *default*, a seguito di una *query*, potrebbero essere presenti righe identiche. Per evitare ciò basta specificare questo facendo seguire il comando `SELECT` dalla clausola `DISTINCT`.

Ridenominazione Per ridenomiare un attributo, basta farlo seguire da `AS Nuovo Nome Attributo`. Mentre per ridenomiare una tabella, basta far seguire il nome originale nella clausola `FROM` dal nuovo nome.

JOIN L'operazione di *Join*, oltre che essere implementato come visto prima tramite la combinazione di `FROM` e `WHERE`, è possibile codificarla in modo esplicito, ecco alcuni esempi:

- `Studenti s JOIN Esami e ON s.Matricola = e.Matricola`
- `Studenti s JOIN Esami e USING Matricola`
- `Studenti s NATURAL JOIN Esami`
- `Studenti s LEFT [OUTER] JOIN Esami e ON s.Matricola = e.Matricola`

Self-Join Per implementare il *self-join* occorre prima ridenominare la tabella due volte:

```
SELECT T1.*, T2.*
FROM Tabella T1, Tabella T2
WHERE T1.codice = T2.codice
```

LIKE Questo operatore è usato per effettuare dei *pattern matching* con una stringa tramite l'uso di due *wildcard*:

- **%** : che denota la presenza di 0 o più caratteri.
- _ : presenza di esattamente 1 carattere.

Nel caso si voglione usare le *wildcards* nel *patter matching*, si imposta un carattere di *escape*:

```
1 SELECT *
2 FROM Modelli
3 WHERE nome_modello LIKE 'C#_F%' ESCAPE '#'
```

NULL Quando si confronta un attributo con `NULL` è sempre consigliabile farlo tramite gli operatori `IS [NOT] NULL`, dato che il confronto tramite l'uguaglianza, nel caso in cui il valore dell'attributo sia `NULL`, restituisce *valore sconosciuto*, ovvero `NULL`, e non un valore *booleano*. Nel calcolo di espressioni, inoltre, se si incontra un valore `NULL`, viene sempre restituito *valore sconosciuto*.

ORDER BY Permette di dare un ordinamento del risultato della `SELECT`, basandosi sui valori di uno o più attributi e opzionalmente indicando in maniera esplicita se l'ordine deve essere crescente (`ASC`, predefinito) o decrescente (`DESC`).

5.1.1 Operatori Aggregati

Nella *target list* possiamo avere anche espressioni che calcolano valori a partire da un insieme di ennuple, e che restituiscono un singolo valore scalare. *SQL* prevede 5 operatori aggregati:

- **Count**: restituisce il numero di righe del risultato della *query*. Mediante la specifica `(*)` si contano tutte le righe selezionate, con `ALL` (specifica di *default*) si contano tutte le righe selezionate che non sono `NULL`, mentre con `DISTINCT` si contano tutte le righe non nulle con valori distinti.
- **Max & Min**: calcolano rispettivamente il massimo e il minimo degli elementi presenti in un colonna dello schema.
- **Avg**: calcola la media dei valori non nulli su una colonna, si possono utilizzare le specifiche `ALL` e `DISTINCT`.
- **Sum**: somma tutti i valori non nulli su una colonna, anche qui si trovano le specifiche `ALL` e `DISTINCT`.

GROUP BY Se si vogliono applicare gli operatori aggregati non sull'intera tabella ma raggruppando i valori per un sottoinsieme di attributi si utilizza la clausola `GROUP BY`, che specifica gli attributi su cui fare i raggruppamenti. Le funzioni di aggregazione saranno applicate su ogni gruppo.

HAVING Tramite questa clausola si possono applicare condizioni sui valori aggregati per ogni gruppo.

SottoSelect o SubQuery È possibile innestare una `SELECT` in un'altra, tramite le parole chiave `ANY`, `ALL`, `[NOT] IN`, `[NOT] EXISTS`, più tutte gli altri operatori della clausola `WHERE`, seguita dalla *sottoselect*.

```
SELECT *
FROM Studenti
WHERE voto > (SELECT AVG(voto)
              FROM Studenti)
```

Oppure è possibile fare operazioni insiemistiche tra i risultati di due `SELECT` tramite gli operatori `UNION`, `INTERSECT`, `EXCEPT`.

Ci sono tre tipologie di *subquery*:

- Subquery Scalari: una `SELECT` che restituisce un solo valore.
- Subquery di Colonna: restituisce una colonna.
- Subquery di Tabella: restituisce una tabella.

Nelle query nidificate le *regole di visibilità* sono simili a quelle dei linguaggi di programmazione, ovvero all'esterno non è possibile riferirsi a variabili definite in *query* più interne o nello stesso livello, viceversa sì.

NOTA Nelle *subquery* non è possibile utilizzare le clausole `HAVING` e `GROUP BY`.

Quantificazione È importante ricordarsi due *"trasformazioni"* molto utili per la quantificazione:

- L'universale negata è uguale all'esistenziale: *non tutti i voti sono* $\leq$ *24* vale a dire *almeno un voto* $>$ *24*.
- L'esistenziale negata è uguale all'universale: *non esiste un voto diverso da 30* vale a dire *tutti i voti sono uguali a 30*.

Quando si utilizza nel `WHERE` un operatore scalare seguito da `ANY` più una *subquery*, il predicato sarà vero solo se almeno uno dei valori restituiti dalla *subquery* lo soddisfa. Mentre se si usa `ALL`, tutti i valori restituiti devono soddisfarlo.

La keyword `EXISTS` invece rappresenta un quantificatore esistenziale, ciò rende vero il predicato se la *subquery* che lo segue restituisce almeno una tupla.
È bene notare che utilizzare `IN` equivale alla forma `=ANY`.

Gli operatori definiti di seguito lavorano su tabelle omogenee, quindi definite sullo stesso dominio.

UNION L'operatore `UNION` prende come operandi i risultati di due `SELECT` e restituisce una terza tabella che contiene l'unione delle righe senza duplicati. Se è scecificata l'opzione `ALL` allora si mantengono anche le righe doppie. Nella tabella risultante i nomi degli attributi vengono presi da quelli del primo operando.

EXCEPT Questo operatore rappresenta la differenza insiemistica. Quindi si mantengono tutte le tuple della prima tabella che non sono presenti anche nella seconda. Anche qui si può utilizzare l'opzione `ALL`.

INTERSECT Questo operatore realizza l'intersezione dell'algebra relazionale, quindi restituisce solo le righe comuni alle due tabelle che restituiscono le due `SELECT`. L'opzione `ALL` è disponibile anche per questo operatore.

5.2 Data Manipulation Language

Il *DML* è il linguaggio di *SQL* per inserire, modificare e cancellare dati nel database e per interrogarlo.

INSERT Per inserire dati nel database si utilizza la seguente sintassi:

```
1 INSERT INTO Nome_Tabella
2 [(Lista_Attributi)] VALUES (Lista_Valori)
```

Le `[]` indicano che specificare la lista degli attributi è opzionale, in caso non è specificata si è obbligati ad assegnare valori a tutti gli attributi nell'ordine opportuno. Se a un attributo non viene assegnato alcun valore, viene assegnato automaticamente `NULL`, a meno che sia stato specificato in precedenza un valore di *default*. Se invece, `NULL` non è permesso per gli attributi mancanti, l'inserimento fallisce.
È possibile effettuare un inserimento prendendo dati da un'altra tabella in questo modo:

```
1 INSERT INTO Tabella_1(Attributo_1, ..., Attributo_n)
2 SELECT Attributo_1, Attributo_n
3 FROM Tabella_2
```

DELETE Per cancellare tuple da una tabella si utilizza la seguente sintassi:

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

Specificando la clausola `WHERE` si vanno ad eliminare solo le tuple che rispettono quella *Condizione*. Nella clausola `WHERE` è possibile utilizzare una *subquery*.

UPDATE Per aggiornare gli attributi di alcune tuple utilizzando il comando `UPDATE`:

```
UPDATE Tabella
SET Nome_Attributo = Espressione
WHERE Condizione
```

5.3 Data Definition Language

Il linguaggio *DDL* permette la creazione, modifica, cancellazione di tabelle, domini e degli altri oggetti del database. Il fine è quello di definire lo *schema logico* del database.

Per la creazione di tabelle si utilizza il comando `CREATE TABLE`:

```
CREATE TABLE Nome_Tabella
(
  Nome_Colonna_1, Tipo_1, Clausola_Default_1, Vincolo_1,
  ...
  Nome_Colonna_n, Tipo_n, Clausola_Default_n, Vincolo_n,
  Vincoli_Tabella
)
```

Questo comando definisce uno schema di relazione e ne crea un'istanza vuota. Successivamente a questo comando, la tabella è pronta per l'inserimento.

DESCRIBE Il comando `DESCRIBE` permette di ottenere lo schema di una tabella dopo che è stata creata.

Un attributo può avere uno dei seguenti tipi:

- CHAR(n): stringhe di lunghezza esattamente n.
- VARCHAR(n): stringhe di lunghezza al più n.
- INTEGER: interi di dimensione uguale alla lunghezza della *word* di memoria dell'elaboratore.

- REAL: numeri reali di dimensione uguale alla lunghezza della *word* di memoria dell'elaboratore.
- NUMBER(p, s): numeri di p cifre di cui s decimali.
- FLOAT(p): numeri binari in virgola mobile con almeno p cifre significative.
- DATE.

ALTER TABLE Con questo comando è possibile modificare lo schema di una tabella:

- ADD (Nome Attributo) per aggiungere una nuova colonna. Se non ci sono altre specifiche l'attributo viene aggiunto come ultima colonna, altrimenti con `FIRST` si aggiunge l'attributo come prima colonna, e con `AFTER Nome Attributo` si aggiunge subito dopo quello indicato.
- DROP (Nome Attributo) per eliminare una colonna. Se un'altra tabella presenta un vincolo di integrità referenziale con l'attributo da eliminare, utilizzando la specifica `RESTRICT`, il comando fallisce; altrimenti utilizzando `CASCADE` la colonna viene eliminata, ed anche tutte le dipendenze logiche (vincolo di *foreign key*) che altre tabelle hanno con questo attributo.
- MODIFY per modificare una colonna.
- SET DEFAULT per assegnare valori di default agli attributi.
- DROP DEFAULT per eliminare l'assegnazione di valori di default.

```
1 ALTER TABLE Tabella
2 ALTER [COLUMN] Colonna
3 SET/DROP DEFAULT
```

- ADD CONSTRAINT per aggiungere vincoli di tabella.
- DROP CONSTRAINT per eliminare vincoli di tabella. Anche qui sono presenti le specifiche `RESTRICT` e `CASCADE`. Infatti con `RESTRICT` non è permesso eliminare i vincoli di unicità e di chiave primaria su una colonna se esistono vincoli di chiave esterna che si riferiscono alla stessa colonna.

Se l'attributo da definire presenta il vincolo `NOT NULL`, occorre prima aggiungerlo senza il vincolo, poi per ogni tupla della tabella si aggiunge un valore per quell'attributo, ed infine si può modificare la tabella impostando quell'attributo come `NOT NULL`.

DROP TABLE Per eliminare una tabella. Utilizzando anche qui `RESTRICT`, l'eliminazione è impedita se la tabella è utilizzata nella definizione di altri oggetti. Con `CASCADE`, invece, vengono eliminate anche tutte le dipendenze che gli altri oggetti hanno con questa tabella.

5.3.1 Vincoli

I vincoli di *integrità* consentono di limitare i valori ammissibile per un'attributo. Quelli *intrarelazionali* (che non si riferiscono ad altre tabelle) sono:

- NOT NULL.
- UNIQUE serve per definire una chiave, e può essere utilizzato sia nella definizione di un attributo, se ne coinvolge solo uno, o per un insieme di attributi, in questo caso viene definito nei vincoli di tabella e si indica la lista di attributi che formano la chiave.
- PRIMARY KEY anche qui può essere definito nella definizione di attributo, se ne coinvolge solo una, o nei vincoli di tabella se la chiave primaria è formata da più attributi. Questo vincolo implica sia il vincolo UNIQUE sull'insieme di attributi, che il vincolo NOT NULL per ogni attributo della chiave primaria.
- CHECK è un'espressione booleana, e richiede che un attributo o un insieme di attributi soddisfino una condizione per ogni tupla della tabella.

I vincoli *interrelazionali*, invece, si impostano per mettere in relazione attributi di tabelle diverse. Queste relazioni sono importanti per eliminare il più possibile la ridondanza dei dati. Per impostare un vincolo di integrità referenziale su un singolo attributo, nella sua descrizione si indica con `REFERENCES Tabella_Riferita(Attributo_Riferito)`. Mentre se il vincolo coinvolge più attributi, anche qui bisogna definirlo nei vincoli di tabella con `FOREIGN KEY (Lista_Attributi_Referenti) REFERENCES Tabella_Riferita(Lista_Attributi_Riferiti)`.

È possibile definire delle politiche di reazione alla violazione quando uno di questi vincoli viene violato medianti i comandi `ON DELETE` e `ON UPDATE`. Le azioni per la `ON DELETE` possono essere le seguenti:

- NO ACTION: questa è l'azione di **default** e blocca la cancellazione delle righe nella tabella riferita.
- CASCADE: cancella tutte le righe dipendenti se si riferiscono ad una riga che viene eliminata dalla tabella riferita.
- SET NULL: assegna `NULL` ai valori delle colonne con il vincolo di *foreign key*, se dipendono da una riga cancellata dalla tabella riferita.

- SET DEFAULT: assegna il valore di *default* ai valori delle colonne con il vincolo di *foreign key*, se dipendono da una riga cancellata dalla tabella riferita.

Le azioni per `ON UPDATE` sono le stesse ma hanno una semantica diversa:

- NO ACTION: rifiuta gli aggiornamenti che violano l'integrità referenziale.
- CASCADE: le righe della tabella referente vengono impostate con i nuovi valori aggiornati della tabella riferita.
- SET NULL: i valori della tabella referente sono impostati a `NULL`.
- SET DEFAULT: i valori della tabella referente sono impostati al valore di *default*.

5.3.2 Viste

Le **viste** sono delle tabelle virtuali i cui dati sono riaggregazioni dei dati contenuti nelle tabelle fisiche del database. Contengono gli stessi dati, ma forniscono una visione diversa e dinamicamente aggiornata.

Permettono di proteggere i database, dato che consentono di limitare l'accesso ai dati, di rappresentare i dati in modo più significativo per l'utente, ma non è possibile utilizzare alcuni operatori come `UNION`, `INTERSECT` ed `EXCEPT`, e neanche la clausola `ORDER BY`.

Il vantaggio principale è che garantiscono l'**indipendenza logica** delle applicazioni rispetto alla struttura del database, cioè è possibile operare modifiche agli schemi nel database senza dover apportare modifiche anche nelle applicazioni che lo usano.

Il comando da utilizzare è il seguente:

```
1 CREATE VIEW Nome_Vista [(Lista_Attributi)] AS Select
2 [with [local | cascaded] check option]
```

Nella lista attributi si trovano i nomi assegnati alle colonne che corrispondono ordinatamente alle colonne selezionate dalla `SELECT`, se sono omesse si utilizzano gli stessi nomi usati nella `SELECT`.

Nota Bene Solo il *contenuto* di una vista è *dinamico*, la sua *struttura* non lo è. Se in seguito alla creazione della vista, alla tabella (o le tabelle) utilizzata nella `SELECT` viene aggiunta una colonna, questa non sarà presente nella vista.

Vista di Gruppo È una vista in cui una delle colonna è una funzione di aggregazione, in questo caso è obbligatorio assegnare un nome alla colonna corrispondente alla funzione. Sono chiamate *viste di gruppo* anche quelle in cui la `SELECT` lavora su altre viste di gruppo.

DROP VIEW Per eliminare una vista si usa il comando:

```
DROP VIEW Nome_Vista {RESTRICT / CASCADE}
```

Con `RESTRICT` la vista viene eliminata solo se non è riferita nella definizione di altri oggetti. Con `CASCADE`, invece vengono eliminate anche tutte le dipendenze da tale vista.

Le tabelle delle viste si possono interrogare come le altre, ma in generale non si possono modificare, a meno che ci sia una corrispondenza biunivoca fra le righe della tabella e quelle della vista, ovvero se nella `SELECT` non sia presente la `DISTINCT` e ci siano solo attributi, nella `FROM` ci deve essere una sola tabella modificabile, il `WHERE` deve essere senza *subquery* e non ci devono essere `GROUP BY` ed `HAVING`.

L'opzione `with check option` assicura che le operazioni di inserimento e modifica dei dati sulla vista soddisfino la clausola `WHERE` nella `SELECT` utilizzata nella creazione della vista.

Nel caso in cui si utilizza la specifica `CASCADED` (di *default*), se la vista è definita in termini di altre viste, allora il DBMS controlla che la nuova tupla soddisfi sia la clausola `WHERE` della vista principale che di quelle di cui è dipendente, indipendemente che quelle viste sono state definiti con l'opzione `with check option`. Invece con la specifica `LOCAL` si verifica solo che soddisfi le clausole delle viste di cui la principale è dipendente con l'opzione `with check option` abilitata.

5.3.3 Procedure e Trigger

Le **procedure** sono delle funzioni che possono essere invocate direttamente dall'utente.

```
CREATE FUNCTION <Nome Funzione> IS
    DECLARE
        <Nome Variabile 1> <Tipo Variabile 1>;
        ...
    BEGIN
        <Query 1>;
        ...
        RETURN(<Nome Variabile>);
    END
```

I **trigger** invece definiscono un'azione che il DBMS deve attivare automaticamente quando si verificano dei determinati *eventi* (cioè quando si eseguono specifici comandi). Questi comandi possono essere principalmente *DML*, quindi `INSERT`, `UPDATE` e `DELETE`, ed anche i comandi per l'aggiornamento di colonne, ma nei DBMS aggiornati si possono utilizzare anche comandi *DDL*.

I *trigger* possono essere eseguiti a livello di riga, quindi imponendo la clausola `FOR EACH ROW` il *trigger* verrà eseguito per ogni riga modifi-

cata dal comando. Mentre se si omette, si parla di trigger a livello di istruzione.

```
CREATE TRIGGER <Nome Trigger>
{BEFORE|AFTER|INSTEAD OF} <Evento 1>, ..., <Evento n>
ON <Tabella Target>
[FOR EACH ROW]
[WHEN <Predicato SQL>]
<Comando SQL o Procedura>
```

Le clausole `BEFORE` e `AFTER` permettono di specificare se le azioni del *trigger* devono essere eseguiti prima o dopo l'esecuzione dei comandi indicati negli eventi.

I *trigger* possono essere classificati come attivi, se modificano lo stato del database, oppure passivi se servono solo a provocare il fallimento dei comandi che si vogliono eseguire sotto certe condizioni.

Ad esempio con la clausola `INSTEAD OF` si può specificare che i comandi che hanno attivato il *trigger* non devono essere eseguiti, e si procederà solo ad eseguire le azioni del *trigger*. I *trigger* di tipo `INSTEAD OF` devono essere sempre a livello di riga e possono anche essere definiti su *viste*.

I *trigger* sono molto utili per aggiunge vincoli che non sono esprimibili nello schema.

5.3.4 Controllo degli Accessi

Ogni risorsa dello schema può essere protetta, il possessore di ogni risorsa assegna privilegi agli altri utenti su quella risorsa. Esiste un utente predefinito chiamato *_system* che rappresenta l'amministratore del *DBMS* ed ha accesso a tutte le risorse. La sintassi per assegnare privilegi è la seguente:

```
GRANT [<Privilegi> | ALL PRIVILEGES]
ON <Risorsa> TO <Utenti>
[WITH GRANT OPTION]
```

I *privilegi* possono essere `SELECT`, `INSERT [(Attributi)]` cioè inserire righe con valori non nulli per gli `Attributi`, `DELETE`, `UPDATE [(Attributi)]` cioè modificare interamente le righe o solo gli `Attributi` per ogni riga, e `REFERENCES [(Attributi)]` ovvero permette di aggiungere vincoli di integrità referenziale sugli `Attributi` in altre tabelle.

L'opzione `WITH GRANT OPTION` permette agli utenti a cui vengono dati i privilegi di trasferirli ad altri.

Per revocare un privilegio:

```
REVOKE <Privilegi> ON <Risorsa> FROM <Utenti>
[RESTRICT | CASCADE]
```

La revoca può essere fatta solo dall'utente che ha assegnato direttamente il privilegio, inoltre con l'opzione `RESTRICT` si specifica che la revoca non deve essere eseguita se questa comporta qualche altra revoca a cascata (nel caso si sia usata l'opzione `WITH GRANT OPTION`); mentre con `CASCADE` si forza la revoca.

Ovviamente chi crea la risorsa ha tutti i permessi su di essa, tranne nel caso delle *viste*, qui il creatore ha gli stessi permessi che ha sulle tabelle usate nella creazione della *vista*.

5.4 Indici e Metadati

Gli **indici** permettono di ordinare i record di una tabella su uno o più attributi in una struttura, in questo modo le ricerche su di essa sono ottimizzate.

```
1 CREATE INDEX <Nome Indice> ON <Nome Tabella>
2 WITH STRUCTURE = BTREE, KEY = (<Attributi>)
```

Metadati Nel database sono presenti delle tabelle che contengono metadati:

- `PASSWORD(username, passowrd)`: tabella delle password.
- `SYSDB(dbname, creator, dbpath, remarks)`: tabella del database.
- `SYSTABLES(name, creator, type, colcount, filename, remarks)`: tabella delle tabelle.
- `SYSCOLUMNS(name, tbname, tbcreator, colno, coltype, lenght, default, remarks)`: tabella delle colonne.
- `SYSINDEXES(name, tbname, creator, uniquerule, colcount)`: tabella degli indici.

6 Normalizzazione

La *teoria della progettazione relazionale* studia le anomalie all'interno degli schemi relazionali e li elimina tramite un processo di **normalizzazione**.

Le *anomalie* possono essere delle ridondanze o delle potenziali inconsistenze quando si effettuano modifiche (anche inserimenti e cancellazioni) nelle istanze.

Definizione (Forma Normale)**.** Una **forma normale** è una proprietà di un database che ne garantisce l'assenza di anomalie.

Per seguire una corretta progettazione si fà in modo che non si uniscano in un'unica relazione attributi che provengano da più classi e associazioni; si cerca di ridurre al minimo la ridondanza; si evita di avere attributi che abbiano frequentemente valori nulli; ed infine bisogna fare in modo che le relazioni possano essere riunite tramite `JOIN` con condizioni di uguaglianze e su attributi che siano o *chiavi primarie* o *chiavi esterne* in modo tale da evitare la generazione di *tuple spurie*.

Definizione (Istanza Valida)**.** Un'**istanza valida** su una relazione è un'istanza a cui viene attribuita anche una nozione semantica, e che quindi è semanticamente corretta nel dominio del discorso.

Definizione (Dipendenza Funzionale)**.** Dato uno schema $R(T)$ e due sottoinsiemi di attributi $X, Y \subseteq T$, una **dipendenza funzionale** fra gli attributi di X e Y è un vincolo espresso nella forma $X \to Y$, e sta a significare che gli attributi di X determinano funzionalmente quelli di Y se per ogni *istanza valida* r su $R(T)$ vale che:

$$\forall\, t_1, t_2 \in r \;.\; t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y]$$

Quando un'istanza r_0 soddisfa una dipendenza funzionale $X \to Y$ si denota con $r_0 \models X \to Y$.

Definizione (Dipendenza Funzionale Atomica)**.** Ogni dipendenza funzionale $X \to A_1, A_2, \ldots, A_n$ si può scomporre nelle **dipendenze funzionali atomiche** $X \to A_1$, $X \to A_2$, $\ldots$, $X \to A_n$.

Definizione (Dipendenza Funzionale Non Banale)**.** Una dipendenza funzionale del tipo $X \to A$ si dice ***non banale*** se $A \not\subseteq X$.

Le dipendenze funzionali possono essere espresse per:

- Espressione Diretta ($P \Rightarrow Q$): $X_= \Rightarrow Y_=$, cioè se in due righe i valori degli attributi in X sono uguali, allora devono esserlo anche in Y.
- Contrapposizione ($\neg Q \Rightarrow \neg P$): $Y_{\neq} \Rightarrow X_{\neq}$, cioè se in due righe i valori degli attributi in Y sono diversi, allora devono esserlo anche in X.

- Per Assurdo: $\neg(X_= \wedge Y_{\neq})$ o $X_= \wedge Y_{\neq} \Rightarrow False$, ovvero non ci possono essere due righe con i valori degli attributi in X uguali e in Y diversi.

Questi sono 3 modi per esprimere la stessa dipendenza funzionale.
La notazione $R < T, F >$ indica una relazione R definita su uno schema di attributi T e dipendenze funzionali F.

Definizione (Dipendenza Funzionale Completa)**.** Una dipendenza funzionale $X \rightarrow Y$ si dice ***completa*** quando per ogni $W \subset X$ non vale $W \rightarrow Y$.

Se X è una *superchiave* allora X determina ogni altro attributo della relazione, quindi $X \rightarrow T$.
Se X è una *chiave* allora $X \rightarrow T$ è una *dipendenza funzionale completa*.

Definizione (Dipendenza Implicata)**.** Sia F un insieme di dipendenze funzionali sullo schema $R(T)$, allora F *implica logicamente* $X \rightarrow Y$ ($F \models X \rightarrow Y$), se ogni istanza di relazione r su $R(T)$ che soddisfa tutte le dipendenze funzionali in F, soddisfa anche $X \rightarrow Y$.

Regole di Inferenza o Assiomi di Armstrong

- Riflessività: $Y \subseteq X \Rightarrow X \rightarrow Y$.
- Arricchimento: $X \rightarrow Y \wedge Z \subseteq T \Rightarrow XZ \rightarrow YZ$.
- Transitività: $X \rightarrow Y \wedge Y \rightarrow Z \Rightarrow X \rightarrow Z$.

Definizione (Derivazione)**.** Dato F un insieme di *dipendenze funzionali*, si dice che $X \rightarrow Y$ è *derivabile* da F (con la notazione $F \vdash X \rightarrow Y$) se $X \rightarrow Y$ può essere inferito da F utilizzando gli *Assiomi di Armstrong*.
Una **derivazione** della dipendenza funzionale f a partire da F è una sequenza finita di dipendenze $f_1, \ldots, f_m$, dove $f_m = f$ ed ogni f_i è o un elemento di F oppure è ottenuta dalle precedenti $f_1, \ldots, f_{n-1}$ dipendenze della derivazione applicando ad una di esse una regola di inferenza.

Usando la *derivazione* si dimostrano le seguenti regole:

- Unione: $\{X \rightarrow Y, X \rightarrow Z\} \vdash X \rightarrow YZ$.
- Decomposizione: Dato $Z \subseteq Y$, allora $\{X \rightarrow Y\} \vdash X \rightarrow Z$.

Teorema (Completezza e Correttezza degli Assiomi di Armstrong)**.** Gli *Assiomi di Armstrong* sono *corretti* e *completi*. Ovvero vale la seguente equivalenza $\models \equiv \vdash$:

- Correttezza: $\forall\ f,\ F \vdash f \Rightarrow F \models f$.
- Completezza: $\forall\ f,\ F \models f \Rightarrow F \vdash f$.

Definizione (Chiusura di F)**.** Dato un insieme F di dipendenze funzionali, la **chiusura** di F denotata con F^+ è:

$$F^+ = \{X \to Y \mid F \vdash X \to Y\}$$

Problema dell'Implicazione Consiste nel decidere se una *dipendenza funzionale* $X \to Y$ appartiene o no ad F^+. Applicando un algoritmo banale, si genera F^+ applicando ad F gli *Assiomi di Armstrong*, ma questo ha ovviamente un ordine di complessità esponenziale.

Definizione (Chiusura di X rispetto ad F)**.** Dato $R < T, F >$ e un sottoinsiemi di attributi $X \subseteq T$, la **chiusura di X rispetto ad F** denotata con X_F^+ è l'insieme:

$$X_F^+ = \{A_i \in T \mid F \vdash X \to A_i\}$$

Teorema. $F \vdash X \to Y \Leftrightarrow Y \subseteq X_F^+$

Algoritmo per Calcolare X_F^+ Si parte da un insieme di attributi X e da un insieme di dipendenze F:

1. Si inizializza X^+ con l'insieme X, dato che per la *riflessività* vale che $X \to X$ indipendentemente da F.
2. Poi, se fra le dipendenze di F c'è una dipendenza $Y \to A$ con $Y \subseteq X^+$ allora si inserisce A in X^+ ($X^+ = X^+ \cup A$). Questo vale dato che se se $Y \subseteq X^+$ allora Y si può derivare funzionalmente da F e per *transitività* si arriva a $Y \to A$.
3. Si ripete il secondo passo finchè non ci sono altri attributi da aggiungere a X^+, quindi abbiamo la certezza che l'algoritmo termini dato che la cardinalità di X è finita.
4. Alla fine $X_F^+ = X^+$.

Definizione (Chiave Candidata)**.** Dato una relazione definita su $R < T, F >$ si dice che un insieme di attributi $W \subseteq T$ è una **chiave candidata** della relazione se:

- $W \to T \in F^+$, ovvero W è una *superchiave*.
- $\forall\, V \subset W,\ V \to T \notin F^+$, ovvero se V è contenuto in W allora V non deve essere una superchiave.

Definizione (Attributo Primo)**.** Un **attributo primo** è un attributo che appartiene ad almeno una chiave.

Definizione (Copertura)**.** Due insiemi di *dipendenze funzionali* sono **equivalenti** ovvero $F \equiv G$, se e solo se $F^+ = G^+$. Si dice anche che F è una **copertura** di G e viceversa.

Definizione (Attributo Estraneo)**.** Dato un insieme di dipendenze funzionali F una dipendenza funzionale $X \to Y \in F$, si dice che X contiene un **attributo estraneo** A_i se e solo se:

$$(X - \{A_i\} \to Y \in F^+)$$

Ovvero $F \vdash (X - \{A_i\}) \to Y$. Per verificare che A_i è estraneo basta calcolare $(X - \{A_i\})^+$ e verificare che includa Y.

In questo caso l'insieme F può essere riscritto eliminando $X \to Y$ e sostituendolo con $(X - \{A_i\}) \to Y$.

Definizione (Dipendenza Ridondante)**.** Dato un insieme di dipendenze funzionali F, si dice $X \to Y$ è una **dipendenza ridondante** se e solo se:

$$(F - \{X \to Y\})^+ = F^+$$

che equivale a dire che $F - \{X \to Y\} \vdash X \to Y$. Per verificare che $X \to A$ è ridondante, la si elimina da F e si calcola X^+, se include A allora la dipendenza è ridondante.

Algoritmo Per Trovare Le Chiavi Ci si basa su due proprietà, se un attributo A non appare a destra di nessuna dipendenza funzionale F allora A appartiene ad ogni chiave della relazione; invece se un attributo B appare solo a destra in tutte le dipendenze funzionali in cui è coinvolto, allora B non appartiene ad alcune chiave.

Detto ciò, si parte dall'insieme di attributi che è presente solo a sinistra delle dipendenze e si calcola la chiusura, se è chiave si termina, altrimenti si aggiunge iterativamente un nuovo attributo all'insieme e si prova a calcolare la chiusura (ovviamente non un attributo che appare solo a destra delle dipendenze).

Definizione (Copertura Canonica)**.** Un insieme di dipendenze funzionali F è detto una **copertura canonica** se e solo se:

- La parte destra di ogni dipendenza funzionale in F è un singolo attributo.
- Non esistono *attributi estranei*.
- Non esistono *dipendenze ridondanti*.

Teorema. Per ogni insieme F esiste una sua *copertura canonica*.

Ridondanze Le *ridondanze* possono essere di due tipi:

- Concettuale: nel database sono memorizzate informazioni che possono essere ricavate da altre già presenti.
- Logica: esistono duplicazioni sui dati che possono generare anomalie, come visto fin qui.

6.1 Decomposizione di Schemi

In linea generale, per eliminare delle *anomalie* da uno schema occorre decomporlo in schemi più piccoli ma che siano equivalenti. L'intuizione è quella di creare uno schema per ogni insieme di dipendenze funzionali che hanno la parte a sinistra uguale e che rappresenti una chiave. Ma questa è una soluzione che non può funzionare sempre.

Definizione (Decomposizione)**.** Data una relazione definita su $R(T)$, $\rho = \{R_1(T_1), \ldots, R_k(T_k)\}$ è una **decomposizione** di R se e solo se $\cup\, T_i = T$. Due proprietà che è desiderabile avere in una *decomposizione* sono:

- La *conservazione dei dati*, intesa come *nozione semantica*.
- E la *conservazione delle dipendenze*.

Definizione (Decomposizione che Preserva i Dati)**.** Data una *decomposizione* $\rho = \{R_1(T_1), \ldots, R_k(T_k)\}$ su $R(T)$, si dice che *preserva i dati* se e solo se per ogni *istanza valida* r su R:

$$r = (\pi_{T_1}(r)) \vee (\pi_{T_2}(r)) \vee \cdots \vee (\pi_{T_k}(r))$$

dove $\vee$ è l'operatore di *natural join*. È importante sottolineare la presenza del simbolo $=$ nell'espressione e non $\subseteq$, dato che non vogliamo la produzione di *ennuple spurie*.

Teorema (Decomposizioni Binarie)**.** Sia $R < T, F >$ uno schema di relazione, la *decomposizione* $\rho = \{R_1(T_1), R_1(T_2)\}$ *preserva i dati* se e solo se:

$$(T_1 \cap T_2 \rightarrow T_1) \in F^+$$

oppure

$$(T_1 \cap T_2 \rightarrow T_2) \in F^+$$

Ciò significa che l'insieme degli attributi comuni alle due relazioni deve essere chiave per almeno una delle due.

Dimostrazione. Si prende un'istanza di relazione r sugli attributi ABC e si considerano le sue proiezioni r_1 su AB e r_2 su AC. Si suppone che A è chiave per r su AC (la dimostrazione è simmetrica per AB) e che quindi r soddisfi la dipendenza funzionale $A \rightarrow C$. Quindi sulla proiezione AC non ci sono tuple diverse quando in A ci sono valori uguali. Se $t = (a, b, c)$ è una tupla del `JOIN` tra r_1 e r_2 si mostra che appartiene sicuramente ad r: t sarà ottenuta dal `JOIN` tra $t_1 = (a, b)$ di r_1 e $t_2 = (a, c)$ di r_2. Quindi per definizione di proiezione in r esisteranno al più due tuple $t_1^{'} = (a, b, *)$ e $t_2^{'} = (a, *, c)$, dove $*$ rappresenta un valore non noto. Poichè $A \rightarrow C$, allora esiste un unico valore di C associato allo stesso valore di A; dato che t_1 e $t_1^{'}$ hanno lo stesso valore di A, il valore non noto $*$ in $t_1^{'}$ deve essere obbligatoriamente c. Quindi $t = t_1^{'}$. □

Anche se una decomposizione è senza perdita può comunque presentare problemi di *conservazione delle dipendenze*. Si dice che una *decomposizione conserva le dipendenze* se ciascuna delle dipendenze funzionali dello schema originario coinvolge attributi che compaiono tutti insieme in uno degli schemi decomposti. Altrimenti, una soluzione è di utilizzare query SQL di verifica (ad esempio facendo uso di *trigger*) per far in modo che le dipendenze vengano soddisfatte lo stesso.

Definizione (Proiezione)**.** Dato lo schema $R < T, F >$ e un sottoinsieme di attributi $T_1 \subseteq T$, la **proiezione** di F su T_1 è:

$$\pi_{T_1}(F) = \{X \to Y \in F^+ \mid XY \subseteq T_1\}$$

Algoritmo per Calcolare $\pi_{T_1}(F)$ L'algoritmo è il seguente:

---

for all $Y \subseteq T_1$ **do**
  $Z \leftarrow Y^+$
  output $Y \to Z \cap T_1$
end for

Definizione (Decomposizione che Preserva le Dipendenze)**.** Dato uno schema $R < T, F >$, la decomposizione $\rho = \{R_1, \ldots, R_n\}$ *preserva le dipendenze* se e solo se l'unione delle dipendenze in $\pi_{T_i}(F)$ è una **copertura** di F.

Il problema di stabilire se una decomposizione preserva o no le dipendenze ha tempo polinomiale.

Teorema. Sia $\rho = R_i < T_i, F_i >$ una decomposizione di $R < T, F >$ che preserva le dipendenze e tale che per qualche j, T_j è una superchiave per $R < T, F >$.

Allora ρ preserva anche i **dati**.

Una decomposizione dovrebbe sempre essere *senza perdita*, quindi in grado di ricostruire tutte le tuple originarie senza generare tuple spurie; e deve *conservare le dipendenze*, in modo da garantire i vincoli di integrità (rappresentate dalle dipendenze funzionali).

6.2 Forme Normali

Definizione (Forma Normale)**.** Una **forma normale** è una proprietà di un database relazionale che ne garantisce l'assenza di *ridondanze* e di comportamenti non deriderabili in fase di aggiornamento.

La prima forma normale **1FN** impone restrizioni sul tipo degli attributi, ovvero ogni attributo deve avere un tipo *elementare*. Mentre la **2FN**, **3FN** e la **FNBC** (*Boyce-Codd*, che è la più restrittiva) impongono restrizioni sulle dipendenze.

6.2.1 BCNF - Forma Normale di Boyce e Cood

Definizione (BCNF)**.** Un'istanza di relazione r definita su $R < T, F >$ è in **BCNF** se e solo se per ogni dipendenza funzionale *non banale* $X \to Y \in F^+$, X contiene una **chiave** K di r, ovvero se X è una *superchiave*.

Infatti se esistesse una dipendenza $X \to Y$ non banale in cui X non è superchiave, vuol dire che X modella un'entità diversa da quella modellata da R, e che quindi è possibile decomporre R.

Algoritmo di Analisi Questo algoritmo riceve in input uno schema $R < T, F >$ con F che è una *copertura canonica* e dà in output una *decomposizione in BCNF* che *preserva i dati* ma non necessariamente le dipendenze. L'algoritmo è il seguente e la sua complessità è esponenziale:

$\rho \leftarrow \{R < T, F >\}$
while Esiste in ρ una $R_i < T_i, F_i >$ non in BCNF per la DF $X \to Y$ **do**
  $T_a \leftarrow X^+$
  $F_a \leftarrow \pi_{T_a}(F_i)$
  $T_b \leftarrow T_i - X^+ + X$
  $F_b \leftarrow \pi_{T_b}(F_i)$
  $\rho \leftarrow \rho - R_i + \{R_a < T_a, F_a >, R_b < T_b, F_b >\}$
end while

6.2.2 3NF

Definizione (3NF)**.** Un'istanza di relazione r definita su $R < T, F >$ è in **3NF** se e solo se per ogni dipendenza funzionale *non banale* $X \to Y \in F^+$, X contiene una **chiave** K di r, oppure ogni attributo in Y è contenuto in almeno una **chiave** K di r. Quindi se X è una *superchiave* oppure se Y è *primo*.

La **3FN** è meno restrittiva della **FNBC**, dato che tollera alcune ridondanze ed anomaile sui dati, quindi certifica uno schema di qualità inferiore. Il vantaggio, però, è che la **3FN** è sempre ottenibile qualunque sia lo schema di partenza.

Algoritmo di Sintesi L'algoritmo prende in input un insieme T di attributi e un insieme F di dipendenze su T. Come output dà una decomposizione $\rho = \{S_i\}_{i=1,\dots,n}$ di T tale che vengano preservati dati e dipendenze e che ogni S_i sia in **3FN** rispetto alle proiezioni di F su S_i. L'algoritmo è il seguente:

1. Si trova una *copertura canonica* G di F e si pone $\rho = \{\}$.

2. Si sostituisce in G ogni insieme di dipendenze con lo stesso determinante X ($X \to A_1, \ldots, X \to A_h$) con la dipendenza $X \to A_1, \ldots, A_h$.
3. Per ogni dipendenza $X \to Y$ in G si aggiunge in ρ uno schema con attributi XY. X diventa la *chiave* della relazione.
4. Si elimina ogni schema di ρ che è contenuto in un qualche altro schema.
5. Se la decomposizione non contiene neanche uno schema i cui attributi costituiscono una superchiave per R, allora si aggiunge a ρ uno schema con attributi W, dove W è una chiave di R.

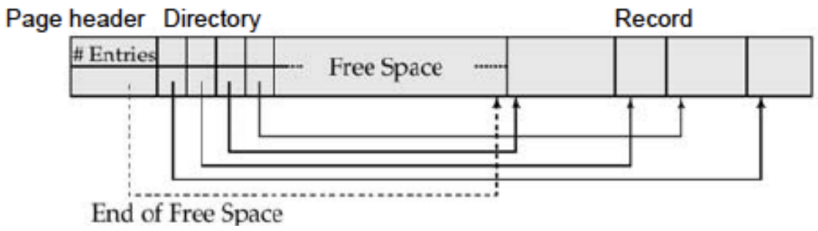
7 Realizzazione DBMS

Un **DBMS** è un sistema software che gestisce grandi quantità di dati, si và incontro a problemi di efficienza, che sono *persistenti*, quindi il sistema deve garantire affidabilità, e *condivisi*, in questo caso deve gestire il controllo degli accessi e della concorrenza. La condivisione dei dati è importante perchè permette di evitare la ridondanza dei dati (che porterebbe ad un spreco di memoria), ma può anche portare a problemi di inconsistenza, dato che se si hanno tante copie di un dato, una modifica di una sola delle copie deve essere propagata nelle altre.

A causa della sua dimensione, un database risiede normalmente sui dischi, e prima che i suoi dati debbano essere elaborati dal DBMS, questi devono essere trasferiti in memoria centrale. Questo trasferimento non coinvolge le singolo tuple ma è effettuato a blocchi o pagine.

Un blocco, fisicamente, è una sequenza continua di settori di una traccia, costituendo l'unità di trasferimento dati da e per la memoria principale. La dimensione tipica è tra i 4 - 64 KB.

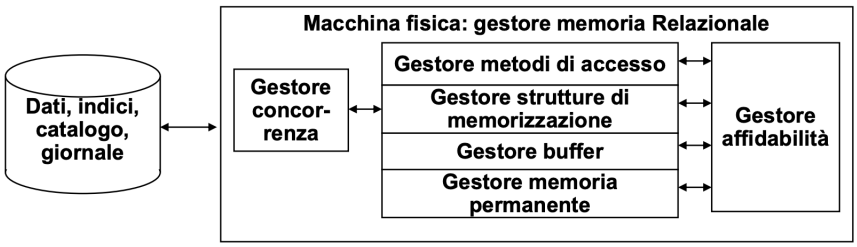
Struttura della Pagina Una pagina, logicamente, è formata da alcune informazioni di servizio e da un'area dove sono contenuti dei *record*. Il riferimento ad un record, chiamato **RID** è formato dalla coppia (**PID** della pagina, posizione del record nella pagina).



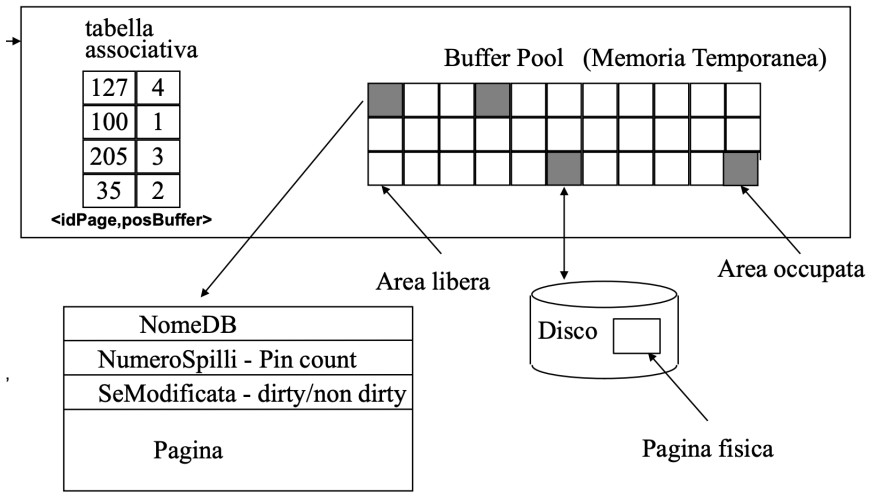
La directory contiene un puntatore per ogni record della pagina, in questo modo è possibile riallocare il record nella pagina senza modificare il **RID**.

7.1 Gestore Memoria

L'architettura a livello fisico è la seguente.



- Gestore Memoria Permanente: fornisce un'astrazione della memoria permanente sotto forma di insiemi di file logici di blocchi fisici, nascondendo le caratteristiche dei dischi e dell'OS.
- Gestore del Buffer: si occupa del trasferimento delle pagine tra la memoria temporanea e quella permanente, offrendo ai livelli superiore una visione della memoria permanente come un insieme di pagine utilizzabili in quella temporanea. In questo modo viene nascosta l'operazione di trasferimento.



Nell'header della pagina presente nella memoria temporanea,

Pin Count viene incrementato se la pagina viene richiesta e decrementato se viene rilasciata. Vengono utilizzate alcune primitive:

- `getAndPinPage(x)`: indica la richiesta al buffer della pagina con `idPage = x` e viene incrementato il `pin count` relativo a quella pagina per indicarne l'uso.
- `unPinPage(x)`: rilascia una pagina e decrementa il *pin count*.
- `setDirty(x)`: imposta il bit `dirty` a 1, che indica se la pagina è stata modificata.
- `flushPage(x)`: forza la scrittura della pagina su disco ed imposta il bit `dirty` a 0.

- Gestore Strutture di Memorizzazione: ogni tipo di organizzazione ha i suoi pro e contro, basandoci sul costo delle operazioni che effettuiamo sui dati e sulla sua occupazione in memoria.
  - Organizzazione Seriale o Heap File: i dati sono memorizzati in modo casuale uno dopo l'altro, il costo in memoria è basso ma è poco efficiente per le operazioni.
  - Organizzazione Sequenziale: i dati vengono ordinati sul valore di uno o più attributi, le inserizioni fanno perdere l'ordinamento, ma le ricerche sono più veloci, dato che si applica la ricerca binaria. Se un file contiene un numero di pagine uguale a `b` allora occorreranno un numero di accessi uguale a $\log_2 b$ per individuare il record.
  - Organizzazioni Per Chiave: in queste organizzazioni, noto il valore di una chiave si trova il record di una tabella con un numero di accessi al disco che dovrebbe essere idealmente uguale ad 1.

7.2 Organizzazioni Per Chiave

7.2.1 Hash File

In un file hash, i record vengono allocati in una pagina il cui indirizzo dipende dal valore di chiave del record stesso. Ad esempio, se un record ha chiave k allora si inserisce nella pagina di indirizzo $H(k)$, dove $H(k) = k \mod NrPage$.

Le collisioni sono gestite con delle *Linked List*. Questa organizzazione è la più efficiente se si considera solo l'accesso diretto basato sui valori della chiave con condizioni di uguaglianza. Non è efficiente invece per ricerche basate su intervalli o su attributi diversi dalla chiave. Inoltre è un'organizzazione procedurale statica, quindi funziona solo su file la cui dimensione varia poco nel tempo.

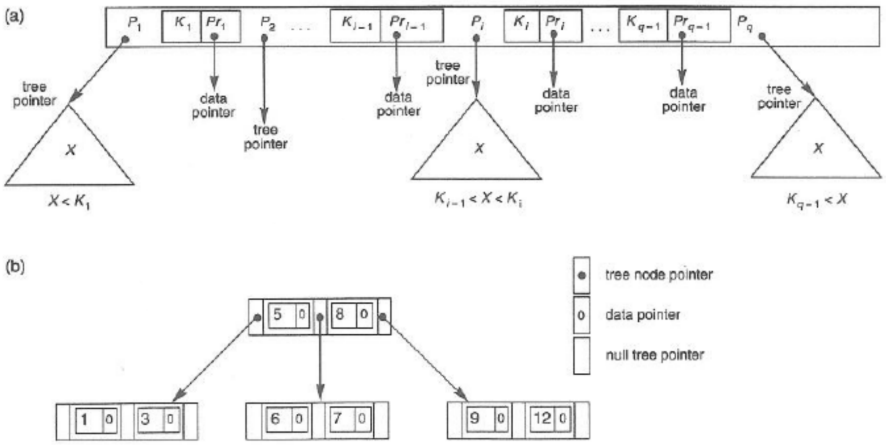
Invece, gli svantaggi che riguardano puramente la struttura dipendono dal numero di pagine, se è troppo piccolo rispetto alla dimensione del database si hanno frequenti collisioni, altrimenti se è troppo grande il fattore di riempimento delle pagine è molto basso.

7.2.2 Metodo Tabellare

Si usa un indice, ovvero un insieme ordinato di coppie `(k, r(k))`, dove `k` è un valore della chiave ed `r(k)` è un riferimento al record con chiave `k`. Gli indici possono essere su più attributi. L'indice viene gestito con una struttura chiamata $B^+ - tree$. Prima di vedere i $B^+ - tree$, vediamo i $B - tree$, dove fissato l'ordine p, questi rappresentano degli alberi di ricerca dove ogni nodo interno ha questa forma:

$$< P_1, < K_1, Pr_1 >, P_2, < K_2, Pr_2 >, \ldots, < K_{q-1}, Pr_{q-1} >, P_q >$$

dove $q \leq p$. I P_i sono chiamati *tree pointer* ovvero rappresentano un puntatore ad un altro node del $B - tree$, K_i è la chiave di ricerca, e Pr_i è un *data pointer*, ovvero è un puntatore ad un record il cui campo *chiave di ricerca* è uguale a K_i oppure ad una pagine che contiene tale record. Per ogni nodo vale la proprietà $K_1 < K_2 < \cdots < K_{q-1}$, e deve avere al massimo p *tree pointer*.



Per tutti i valori X della chiave di ricerca che appartengono al sottoalbero puntato da P_i vale che:

$$K_{i-1} < X < K_i \ \ per\ 1 < i < q$$

$$X < K_i \ \ per\ i = 1$$

$$K_{i-1} < X \ \ per\ i = q$$

I vincoli che il $B - tree$ deve rispettare, che lo differenziano da un normale albero di ricerca sono:

- La radice ha almeno due *tree pointer* a meno che non sia l'unico nodo.
- Ogni nodo, esclusa la radice, ha almeno $\lceil \frac{p}{2} \rceil$ *tree pointer*.
- Tutti i nodi foglia sono posti sullo stesso livello.

Questi vincoli garantiscono che il $B - tree$ sia bilanciato.

I $B^+ - tree$, sono essenzialmente dei $B - tree$ dove i *data pointer* sono memorizzati esclusivamente nei nodi foglia dell'albero. Quindi, fissato l'ordine p, ogni nodo interno ha la seguente forma:

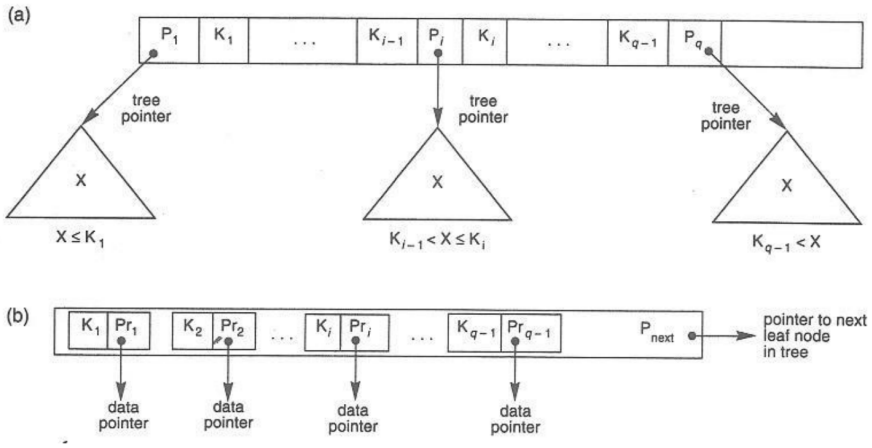
$$< P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q >$$

con $q \leq p$. Un'altra piccola differenza è che per ogni valore X della chiave di ricerca puntata da P_i si ha che:

$$X \leq K_i \ \ per\ i = 1$$

$$K_{i-1} < X \leq K_i \ \ per\ 1 < i < q$$

$$K_{i-1} < X \ \ per\ i = q$$



La struttura dei nodi foglia è la seguente:

$$<< K_1, Pr_1 >, < K_2, Pr_2 >, \ldots, < K_q, Pr_q > P_{next} >$$

dove $q \leq p_{leaf}$ e dove vale sempre $K_1 < K_2 < \cdots < K_q$. P_{next} è un *tree pointer* che punta al successivo nodo foglia del $B^+ - tree$. Ogni Pr_i,

invece, è un *data pointer* che punta al record con campo di ricerca uguale a K_i, a un blocco contenente tale record, o a un blocco di puntatori ai record con campo di ricerca uguale a K_i, se il campo di ricerca non è una chiave. Ogni nodo foglia ha almeno $\lceil \frac{p_{leaf}}{2} \rceil$ valori.

Ovviamente è importante sottolineare che gli alberi stessi sono memorizzati su disco, assegnando ad ogni nodo una pagina.

Esistono due tipologie di indici ad albero:

- Indici Statici: la struttura ad albero viene creata sulla base dei dati presenti nel database e non viene più modificata se non periodicamente.
- Indici Dinamici: la struttura ad albero viene aggiornata ad ogni operazione di inserimento e cancellazione.

Gli indici si possono anche distinguere in:

- Indici Primari: in questo caso la chiave di ordinamento del file sequenziale coincide con la chiave di ricerca dell'indice. È costituito da un insieme di coppie `<K(i), RID(i)>`, dove il primo campo è la chiave primaria ed è dello stesso tipo del campo chiave di ordinamento, mentre il secondo campo è un puntatore ad un blocco del disco.
- Indici Secondari: qui invece la chiave di ordinamento e quella di ricerca sono diverse. Questo indice può essere definito su un campo che non è chiave ma è chiave candidata, quindi ha valori univoci, oppure su un campo che ha valori duplicati. In questo caso il primo campo della coppia è chiamato campo di indicizzazione.

7.3 Ordinamento

L'ordinamento delle tabelle è importante non solo per ottenere il risultato di un `ORDER BY`, ma anche in altre operazioni relazionali come la `JOIN` e la `GROUP BY` per ottimizzare i tempi.

L'algoritmo più usato dai *DBMS* è il **Merge Sort a Z vie**, dove a *Z vie* indica che l'operazione di merge viene eseguita su *Z* pagine alla volta. Se si suppone di avere un file di NP pagine e di avere un numero di buffer $NB < NP$ in memoria centrale; l'algoritmo opera in due step:

- Prima esegue un sort interno, utilizzando ad esempio il *Quicksort*, in cui all'interno di ogni pagina i record vengono ordinati, le pagine ordinate vengono chiamate *run*.
- Infine, operando uno o più passi di fusione, in base al numero di vie usate, le *run* vengono fuse fino a produrne una sola.

Supponendo di avere a diposizione $NB = 3$ buffer, dove due sono quelli di input e uno è quello di output, e di considerare nel calcolo della complessità solo il numero di operazioni *I/O* effettuate tra memoria centrale e fisica.

Nella fase di sort interno, si scriverà ogni pagina in uno dei buffer di input e poi si riscriverà ordinata in quello di output, impiegando un numero di operazioni uguale a $2 \cdot NP$. Ad ogni passo di merge in totale si leggeranno e scriveranno sempre $2 \cdot NP$ pagine (i record bisogna visitarli sempre tutti), e il numero di passi di merge sarà $\lceil \log_2 NP \rceil$ in quanto ad ogni passo il numero di *run* si dimezza. Il costo totale è quindi pari a $2 \cdot NP \cdot (\lceil \log_2 NP \rceil + 1)$.

Si può osservare che avendo a disposizione NB buffer si possono ordinare NP pagine alla volta nella fase di sort interno anzichè una sola, il che porta ad un costo di $2 \cdot NP \cdot (\lceil \log_2 \frac{NP}{NB} \rceil + 1)$.

Inoltre se abbiamo $NB > 3$ buffer si possono fondere $NB - 1$ run alla volta (1 buffer è per l'output). In questo caso il costo è $2 \cdot NP \cdot (\lceil \log_{NB-1} \frac{NP}{NB} \rceil + 1)$.

7.3.1 JOIN

Il costo di una `JOIN` può essere molto elevato, dato che il prodotto cartesiano tra due tabelle è molto grande, e questo seguito da una restrizione può risultare inefficiente. Inoltre, anche se `JOIN` è commutativo, la scelta tra l'operatore da posizionare a sinistra e tra quello da posizionare a destra è cruciale dal punto di vista dell'ottimizzazione computazionale.

Esistono 4 algoritmi per l'implementazione del `JOIN`, chiameremo con R la relazione a sinistra (anche detta esterna), e con S la relazione a destra (interna):

- Nested Loops: qui il costo dipende dallo spazio a disposizione in memoria centrale, nel caso si ha solo un buffer per R e uno per S, bisogna caricare ogni pagina di R e per ogni record della pagina si carica ogni volta una pagina di S, quindi il numero di operazioni di *I/O* diventa $NPage(R) + NRec(R) \cdot NPag(S)$. Nel caso è possibile allocare un numero di buffer per S uguale a $NPag(S)$, il costo diventa $NPag(R) + NPag(S)$. La prima formula si può anche riscrivere in questo modo:

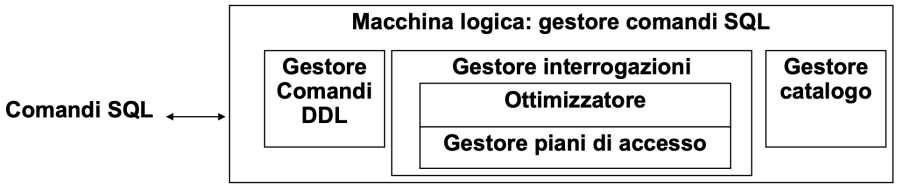
  $$NPag(R) \cdot \frac{NRec(R)}{NPag(R)} \cdot NPag(S)$$

  che equivale a dire che per ogni pagina di R, scriviamo nel buffer di S tutte le sue pagine tante volte quanti sono i record per ogni pagina di R. È semanticamente la stessa cosa. Dato che l'unico fattore che può cambiare in base a se si sceglie R o S a sinistra

è $\frac{NRec(X)}{NPag(X)}$, si prende R a sinistra se $\frac{NRec(R)}{NPag(R)} < \frac{NRec(S)}{NPag(S)}$, altrimenti si prende S. Ciò sta a significare che in R, per ogni pagina ci sono meno record, e che quindi le tuple di R sono più grandi di quelle di S.

- Page Nested Loops: questa può essere vista come una variante del *Nested Loops*, in quanto quando carichiamo nel buffer di R una sua pagina poi non si vanno a prendere tutte le pagine di S tante volte quante sono i record della pagina di R, ma, quando si carica la pagina di S si esegue il *JOIN* di ogni record nel buffer di R con ogni record del buffer di S. L'unico compromesso è che nella tabella risultante, non viene preseservato l'ordine della relazione esterna. Il costo qui è $NPage(R) + NPage(R) \cdot NPage(S)$.
- Index Nested Loops: in questo caso la scansione completa di S viene sostituita da una scansione basata su un *indice* costruito sugli attributi utilizzati per la `JOIN`.
- Merge-Sort: questo algortimo è applicabile quando entrambe le tabelle sono ordinate sugli attributi di `JOIN`. La logica dell'algortimo sfrutta il fatto che entrambi gli input sono ordinati, in questo modo vengono evitati confronti inutili ed il costo diventa $NPag(R) + NPag(S)$.

7.4 Piani di Accesso



L'*ottimizzatore* sceglie, in base al comando *SQL*, il piano d'accesso dal costo minimo, usando le statistiche presenti nel *catalogo*. Nel processo di ottimizzazione, dati il comando *SQL* e il catalogo, viene prima effettuata una procedura di analisi lessicale e semantica, e di semplificazione del comando, questa fase dà in output un albero di operatori logici dell'algebra relazionale, sul quale si esegue una trasformazione utilizzando delle regole di equivalenza. Successivamente si applica un'ottimizzazione fisica dove gli operatori logici sono trasformati in operatori fisici e la loro scelta definisce il piano d'accesso. In quest'albero, le foglie sono le tabelle e i nodi interni specificano le modalità con cui gli accessi e le operazioni relazionali sono effettuate.

Definizione (Piano d'Accesso)**.** Scelta dell'algoritmo per eseguire ogni operatore.

Infatti ogni operatore logico può essere realizzato mediante più algoritmi che rappresentano gli operatori fisici.

Ogni operatore fisico è un'*iteratore*, ovvero un oggetto con i seguenti metodi (realizzati dalla macchina fisica):

- `open()`: inizializza lo stato dell'operatore allocando i buffer per gli input e gli output, inoltre richiama ricorsivamente la `open()` sugli operatori figli e passa anche i parametri.
- `next()`: ritorna la prossima tupla del risultato, include anche la `next()` sui figli.
- `isDone()`: indica se ci sono ancora tuple da leggere.
- `reset()`.
- `close()`: termina l'esecuzione dell'operatore e rilascia le risorse ad esso allocate.

Ecco una lista di operatori logici ed i corrispondenti fisici:

- **R**:
  - TableScan(R): per la scansione della relazione R.
  - IndexScan(R, Idx): scansione di R utilizzando l'indice Idx.
  - SortScan(R, $\{A_i\}$): scansione di R ordinata sull'insieme $\{A_i\}$.
- π_{A_i}:
  - Project(O, $\{A_i\}$): per la proiezione dei record di O senza eliminare i duplicati.
  - Distinct(O): per eliminare i duplicati dei record ordinati di O.
- σ_Ψ:
  - Filter(O, Ψ): per la restrizione dei record di O senza indici.
  - IndexFilter(R, Idx, Ψ): per la restrizione con indice dei record di R. Ovviamente, gli attributi utilizzati nella condizione Ψ devono essere gli stessi presenti nell'indice Idx.
- $\tau\{A_i\}$:
  - Sort(O, $\{A_i\}$): per ordinare i record di O per valori crescenti degli $\{A_i\}$.
- $\{A_i\}\gamma\{f_i\}$:

  - GroupBy(O, $\{A_i\}$, $\{f_i\}$): per raggruppare i record di O sugli attributi $\{A_i\}$ e usando le funzioni di aggregazione in $\{f_i\}$. In $\{f_i\}$ vi sono le funzioni presenti nella `SELECT` e nella `HAVING`. I record di O devono essere ordinati sugli $\{A_i\}$.

- $\bowtie_{\Psi_J}$:
  - NestedLoop(O_E, O_I, Ψ_J): in questo caso, semplicemente, il *NestedLoop* invoca la `next()` prima su O_E e poi su O_I, poi esegue la condizione e se è rispettata aggiunge la nuova tupla al risultato, in seguito ripete questa procedura iterativamente con le `next()` finchè non ci sono più tuple di O_E e O_I.
  - PageNestedLoop(O_E, O_I, Ψ_J).
  - IndexNestedLoop(O_E, O_I, Ψ_J): L'operando interno (O_I), deve essere un *IndexFilter*(R, Idx, $\Psi_J^{'}$) oppure dev'essere un *Filter*(O, $\Psi^{''}$) dove O è comunque un *IndexFilter*(R, Idx, $\Psi_J^{'}$). La condizione $\Psi_J^{'}$ è uguale alla condizione di giunzione Ψ_J, solo che a runtime gli attributi di O_E sono sostituiti dai valori presenti per ogni tupla r di O_E.

    A livello procedurale, qui è un po' diverso rispetto al *Nested Loop*. La *IndexNestedLoop*, invoca prima la `open()` su O_E e per ogni tupla restituita dalla `next()`, utilizza la `open()` su O_I passandogli come parametri i valori degli attributi di O_E presenti nella condizione di `JOIN`, cosicchè l'*IndexFilter* presente in uno dei livelli inferiori possa restituire le tuple che rispettano la condizione (che come detto prima cambia a runtime); successivamente l'*IndexNestedLoop* chiamarà iterativamente la `next()` su O_I per ottenere queste tuple.
  - SortMerge(O_E, O_I, Ψ_J).

È importante sottolineare che gli operatori che presentano R sono quelli che operano direttamente sulle tabelle, e che quindi nell'albero sono sempre delle foglie, mentre gli operatori che presentano O operano su un oggetto che è il risultato di un altro nodo nel livello inferiore.

8 Gestione delle Transazioni

Le **transazioni** rappresentano l'unità di lavoro elementare (costituita da un insieme di istruzioni SQL) che modificano il contenuto di un database.

Le *transazioni* hanno questa sintassi:

```
1  begin transaction
2  [Istruzioni SQL]
3  commit work
```

```
1  begin transaction
2  [Istruzioni SQL]
3  if [Condizione] commit work;
4  else rollback work
```

Proprietà delle Transazioni

- Atomicity: le transazioni deve essere eseguite in modo tale che i comandi SQL all'interno di esse vengano o eseguiti tutti o non eseguiti proprio.

  Quindi le transazioni che terminano prematuramente sono trattate dal sistema come se non fossaro mai iniziate, quindi eventuali loro effetti sul database vengono annullati.

- Consistency: le transazioni devono lasciare il database in uno stato *consistente*, ovvero in uno stato dove i vincoli di integrità non devono essere violati.

- Isolation: l'esecuzione di una transazione deve essere indipendente dalle altre.

  Quindi nel caso di esecuzioni concorrenti di più transazioni, il loro effetto complessivo è quello di un'esecuzione seriale.

- Durability: l'effetto di una transazione che ha eseguito un `commit work` non deve andare perso.

  Quindi le modifiche sul database di una transazione terminata con successo sono permanenti, ovvero non possono essere alterate da eventuali malfunzionamenti.

La *Gestione delle Transazioni* comprende la *Gestione dell'Affidabilità* e la *Gestione della Concorrenza*.

8.1 Gestione dell'Affidabilità

Definizione (Transazione)**.** È un'unità logica di elaborazione che corrisponde ad una serie di operazioni fisiche elementari sul database.

Una **transazione** viene anche vista come una sequenza di azioni di lettura e scrittura in memoria permanente e di azioni di elaborazione dati iin memoria temporanea.

Il **Gestore dell'Affidabilità** garantisce le proprietà di *Atomicity* e di *Durability*, è responsabile dell'implementazione dei comandi `begin transaction`, `commit` e `rollback` e di ripristinare il sistema dopo malfunzionamenti software (*ripresa a caldo*) e hardware (*a freddo*).

Per aumentare l'efficienza, tutti i *DBMS* dispongono di un buffer temporaneo di dati in memoria principale, il quale viene periodicamente scritto in memoria secondaria.

Le transazioni che interessano a noi sono solo quelle di lettura e scrittura indicate rispettivamente con $r_i[x]$ e $w_i[x]$. Il singolo dato coinvolto può essere un *record*, un *campo* di un record o una *pagina*, per semplicità si tratterà solo di pagine.

L'operazione $r_i[x]$ comporta la lettura di una pagine nel buffer se non è già presente.

L'operazione $w_i[x]$, invece, comporta le lettura nel buffer di una pagina se non è già presente, e la sua modifica, ma non necessariamente la sua scrittura in memoria permanente. Questo è il motivo per cui in caso di malfunzionamenti si potrebbe perdere l'effetto di un operazione.

Tipi di Malfunzionamento

- Fallimenti di transazioni: non comportano la perdita di dati in memoria permanente o temporanea, ma sono dovuti a violazioni di vincoli di integrità e di protezione.
- Fallimenti di sistema: comportano la perdita di dati in memoria temporanea ma non permanente.
- Disastri: comportano perdita di dati in memoria permanente, può essere dovuto per esempio al danneggiamento della periferica.

Il controllore di affidabilità utilizza un file di log nel quale sono indicate tutte le operazioni svolte dal *DBMS*.

Data una transazione T, con **B(T)** si indica il record di ***begin***, con **C(T)** quello di ***commit*** e con **A(T)** quello di ***abort*** relativi a T. Mentre con **U(T, O, BS, AS)**, **I(T, O, AS)** e **D(T, O, BS)** si indicano i record di ***UPDATE***, ***INSERT*** e ***DELETE***. Dove O è l'oggetto su cui sono eseguite le operazioni, mentre BS indica lo stato precedente, e AS quello successivo.

Questi record permettono di *disfare* e *rifare* le corrispettive azioni sul database. Le primitive di **undo**, disfano un'azione sull'oggetto O, ricopiando in O il valore BS.

Mentre le primitiva di **redo**, si utilizzano per rifare un'azione su un oggetto O, ovvero si ricopia in O il valore di AS.

I record possono essere di due tipi:

- **Record di Transazione**: quelli visti prima.

- **Record di Sistema**: tengono traccia delle operazioni di sistema, e possono essere di due tipi, *dump* e *checkpoint*. L'operazione di *dump* produce una copia completa del database, ed è effettuata in mutua esclusione con tutte le altre transazione, quando il sistema non è operativo. La copia di *backup* viene salvata in memoria permanente.

Le regole di scrittura sul log sono due:

- Write Ahead Log (WAL): la parte BS di ogni record deve essere scritta sul log prima che l'operazione venga effettuata sul database.
- Commit Precedence: la parte AS di ogni record deve essere scritta sul log prima di effettuare il `commit` della transazione.

Gli algoritmi si differenziano in base a come trattano le transazioni e le scritture sul database. Noi lavoriamo sull'ipotesi che le scritture sul log vengono portate immediatamente nella memoria permanente.

Le modifiche al database si portano mediante la politica della *modifica libera*, ovvero le modifiche possono essere portare al database prima che la transazione termini.

La regola per disfare, invece si chiama **Log Ahead Rule**: se una nuova versione di una pagina rimpiazza la vecchia sul database prima che la transazione esegua il `commit`, allora la vecchia versione deve essere prima scritta sul log in modo permanente.

La terminazione di una transazione invece si gestisce con la regola del *commit libero*, ovvero una transazione può essere considerata terminata normalmente prima che tutte le modifiche vengano riportate sul database.

Invece la regola per rifare una transazione si chiama **Commit Rule**, cioè le nuove versione delle pagina di una transazione devono essere portate in modo stabile nel log prima che la transazione raggiunga il commit.

Quando si verifica un ripristino, solo gli aggiornamenti più recenti tra quelli sul log potrebbero non essere stati riportati sul database. Per avere la certezza che alcune operazioni non debbano essere rieseguite, ci fà periodicamente un checkpoint, ovvero si scrive la marca *CKP* sul log per indicare che tutte le operazioni che la precedono sono state effettuate sul database.

Quando si vuole fare un checkpoint, si scrive sul log una marca di inizio checkpoint che riporta un elenco di tutte le transazioni attive, `BeginCkp{T1, ..., Tn}`, in seguito, in parallelo a tutte le altre operazioni delle transazioni, il gestore del buffer riporta sul disco tutte le pagine modificate e alla fine si scrive sul log la marca `EndCkp` che certifica che tutte le scritture avvenute prima di `BeginCkp` sono sul disco, mentre quelle fra `BeginCkp` e `EndCkp` potrebbero esserci sul disco come potrebbero non esserci.

Gestione dei Fallimenti

- Fallimenti di Transazioni: si scrive sul log il record `(T, abort)` e si applica la procedura di *disfare*.
- Fallimenti di Sistema: il database viene ripristinato col comando `Restart` avendo come riferimento l'ultimo checkpoint e procedendo in modo che le transazioni non terminate vengano disfatte e quelle terminate vengano rifatte. Si utilizza una *ripresa a caldo*, ovvero si trova l'ultimo checkpoint ripercorrendo il log a ritroso costruendo gli insiemi $UNDO$ e $REDO$ con le transazioni da disfare e rifare. Si ripercorre di nuovo il log all'indietro fino all'ultima azione delle transazioni in $UNDO$ disfacendole tutte. Infine si ripercorre il log in avanti rifacendo tutte le transazioni in $REDO$.
- Disastri: qui si riporta in linea l'ultimo backup del database e lo si aggiorna rifacendo le modifiche delle transazioni terminate normalmente. Qui invece si applica una *ripresa a freddo* eseguendo le operazioni descritte in precedenza e si segue con una *ripresa a caldo*.

8.2 Gestione della Concorrenza

Uno *schedule* S si dice ***seriale*** se le azioni di ciascuna transazione appaiono in sequenza senza essere spezzate da azioni di altre transazioni.

$$S = \{T_1, T_2, \ldots, T_n\}$$

Uno *schedule seriale* è ottenibile se le transazioni sono eseguite una alla volta o se sono completamente indipendenti una dall'altra, ovvero l'intersezione dell'insieme di dati che accedono è sempre vuota. Ma questi sono scenari che non sono per niente realistici.

Di solito, un *DBMS* tradizionale gestisce questi problemi garantendo una proprietà di *isolamento*, ovvero facendo in modo che ogni transazione sia eseguita come se non ci fosse concorrenza. Nella realtà, però, la concorrenza permette di raggiungere maggiori livelli di efficienza e scalabilità. Questo, tuttavia comporta la gestione di alcune anomalie, come la *perdita di aggiornamento* e la *lettura sporca* o *inconsistente*.

Definizione (Esecuzione Seriale)**.** Un'esecuzione di un insieme di transazioni $\{T_1, \ldots, T_n\}$ si dice ***seriale*** se per ogni coppia di transazioni T_i e T_j, tutte le operazioni di T_i vengono eseguite prima di qualsiasi operazione di T_j o viceversa.

Definizione (Esecuzione Serializzabile)**.** Un'esecuzione di un insieme di transazioni si dice ***serializzabile***, se produce lo stesso effetto sul database di quello ottenibile eseguendo serialmente in qualche ordine solo le transazioni terminate normalmente.

Per garantire la *serializzabilità* delle transazioni concorrenti ci sono due classi di protocolli:

- Protocolli Pessimistici: tendono a ritardare l'esecuzione di transazioni che potrebbero generare conflitti e anomalie, rispetto alla transazione eseguita in quell'istante. L'approccio quindi è quello di prevenire.
- Protocolli Ottimistici: questi permettono l'esecuzione sovrapposta e non sincronizzata delle transazioni e si effettuano controlli sui possibili conflitti generati solo quando si è in procinto di eseguire il `commit`. Se i controlli rilevano dei conflitti viene effettuato il `rollback` delle azioni delle transazioni e la loro relativa riesecuzione. I protocolli di questo tipo su basano su tre fasi:
  - Fase di Lettura: ogni transazione legge prima i valori degli oggetti su cui deve operare dal database e li copia in variabili locali dove sono eseguiti gli aggiornamenti.
  - Fase di Validazione: si effettuano dei controlli sulla serializzabilità nel caso in cui gli aggiornamenti locali devono essere propagati nel database.
  - Fase di Scrittura: nel caso in cui gli oggetti aggiornati superano la fase di validazione, si scrivono sul database.

Per implementare il controllo della concorrenza, i *DBMS* possono utilizzare dei meccanismi di `lock`, ovvero per ogni risorsa esiste una `lock` di *lettura* che permette l'accesso condiviso, ed una di *scrittura* che invece è in mutua esclusione. Le `lock` utilizzate si dicono *multi granulari* dato che possono lavorare sia a livello di tabella, che di riga e di pagina.

- $lock_r(x)$: richiesta del `lock` in lettura su x.
- $lock_w(x)$: richiesta del `lock` in scrittura su x.
- $unlock(x)$: rilascio del `lock` su x.

Il gestore della concorrenza, chiamato **serializzatore**, ha il compito di stabilire l'ordine in cui devono essere eseguite le singole operazioni per rendere serializzabile l'esecuzione di un insieme di transazioni.

Definizione (Strict Two Phase Locking)**.** Questo protocollo è definito da tre regole:

- Ogni transazione prima di effettuare un'operazione deve acquisire il `lock` corrispondente.
- Transazioni diverse non possono ottenere `lock` in conflitto.

- Le `lock` non vengono rilasciate fino al `commit` o `abort` della transazione.

Questo tipo di protocollo può però generare situazioni *deadlock*. Per gestirle si possono usare tre tecniche:

- Uso dei **Timeout**: ovvero ad ogni operazione di `lock` ha associato ad essa un *timeout* entro il quale deve essere completata, pena l'`abort` della transazione.
- **Deadlock Avoidance**: per prevenire le configurazioni che potrebbero portare ad un *deadlock* si cerca di fare il `lock` e `unlock` delle risorse nello stesso momento, oppure l'utilizzo di *timestamp* o di *classi di priorità* tra le transazioni, anche se queste ultime due tecniche potrebbe determinare delle situazioni di *starvation*.
- **Deadlock Detection**: si utilizzano algortimi per capire se ci si trova in un *deadlock* e gestire la situazione. Ad esempio, si può usare un grafo di richieste e risorse, e la presenza di un ciclo corrisponde ad un *deadlock*. Quindi nel caso di un ciclo si fa l'`abort` di tutte le transazioni coinvolte.

Un altro metodo per la gestione della concorrenza è l'utilizzo dei **timestamp** delle transazioni, ovvero ad ogni transazione si associa un *timestamp* che rappresenta il momento in cui inizia la transazione e ogni transazione non può leggere o scrivere un dato scritto da una transazione con *timestamp* maggiore, e non può scrivere su un dato già letto da una transazione con *timestamp* maggiore.

Livelli di Isolamento

- SERIALIZABLE: assicura che una transazione T legge solo i cambiamenti fatii da transazioni già concluse; nessun valore letto o scritto da T verrà cambiato da altre transazioni finchè T non è conclusa; e se T legge un insieme di valori acceduti secondo una certa condizione di ricerca, l'insieme non viene modificato da altre transazioni finchè T non è conclusa.
- REPEATABLE READ: assicura che la transazione T legge solo cambiamenti fatte da transazioni concluse e che nessun valore letto o scritto da T verrà cambiato da altre transazioni finchè T non è conclusa.
- READ COMMITTED: assicura che la transazione T legge solo cambiamenti fatti da transazioni concluse e che T non vede nessun cambiamento tra i valori letti all'inizio di T, effettuato da transazioni concorrenti. non concluse

- READ UNCOMMITTED: una transizione T può leggere modifiche su un oggetto fatte da una transazione in esecuzione, e dato che l'oggetto può essere cambiato mentre T è in esecuzione, si dice che T è soggetta ad effetti fantasma.