Base di dati

Ahmad Shatti

2020-2021

Indice

Chapter 1

Sistemi per basi di dati

1.1 Sistemi informativi e informatici

Una **base di dati** è una tecnologia di base, introdotta negli anni '70, che gestisce le attività quotidiane dell'azienda e viene utilizzata anche per svolgere un'attività di analisi dell'andamento dell'organizzazione. Con il tempo le attività di analisi sono state gestite ponendo le interrogazioni non direttamente sulla base di dati ma costruendo un secondo strumento, il **Data Warehouse**, per l'esecuzione di attività analitiche. Data Warehouse, Data Lake, Big Data e Data science sono termini che hanno a che vedere con l'analisi dei dati e non rientrano tra i temi di questo corso.

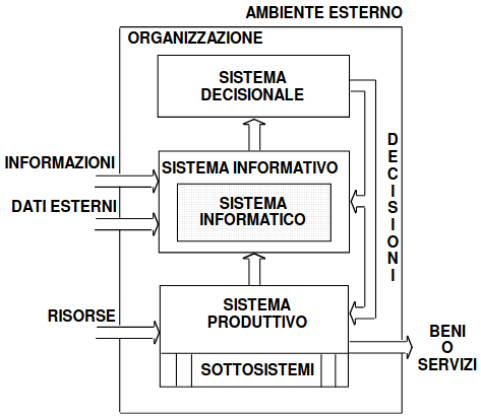
Studenti

Nome	Matricola	Provincia	Nascita
Isaia	71523	Pisa	1980
Rossi	67459	Lucca	1981
Bianchi	79856	Livorno	1980
Bonini	75649	Pisa	1981

ProveEsami

Materia	Candidato	Data	Voto
BD	71523	12/01/01	28
BD	67459	15/09/02	30
IA	79856	25/10/03	30
BD	75649	27/06/01	25
IS	71523	10/10/02	18

Una base di dati è una insieme di dati strutturati raccolte in più collezioni in relazione fra di loro, la si può immaginare come una serie di tante tabelle in relazione fra di loro. Un **sistema informativo** non fa confuso con la sua base di dati perchè è uno strumento molto più ricco: è una combinazione di risorse, umane e materiali, e di procedure organizzate per *raccolta*, *archiviazione*, *elaborazione* e *scambio* delle informazioni necessarie alle attività *operative*, di *programmazione e controllo*, e di *pianificazione strategica*. All'interno del sistema informativo vi è il **sistema informatico** che ha un ruolo cruciale e possiede insieme alla base di dati altri strumenti importanti come il *software* e l'*hardware di base*, uno *schema* che descrive la struttura della base di dati, i *programmi applicativi* che forniscono i servizi agli utenti eseguendo un certo insieme di operazioni sulla base di dati e la *comunicazione* che permette l'accesso ai servizi del sistema informatico ad utenti e programmi.

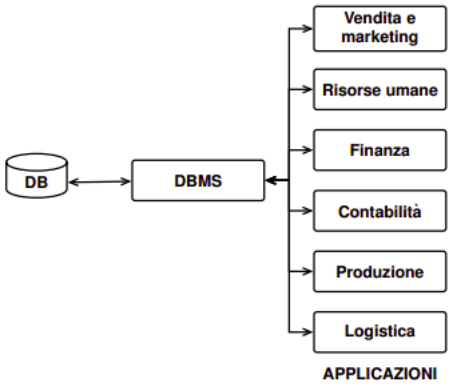


1.2 Tipi di sistemi informatici

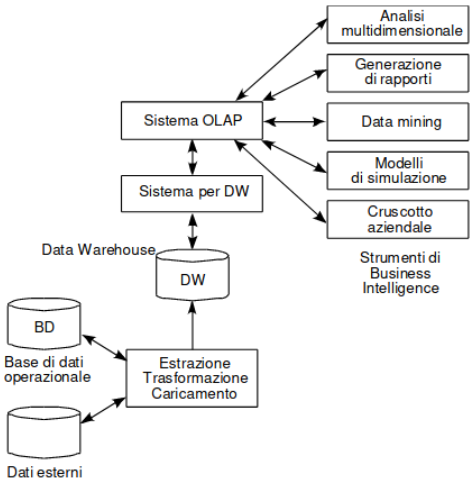
I sistemi informatici vengono distinti in due categorie, dal punto di vista di chi li usa:

- **Sistemi informatici operativi** si usano per svolgere le classiche attività strutturate e ripetitive dell'azienda e vengono utilizzate basi di dati. Il **DBMS** è lo strumento che supporta la base di dati e che viene utilizzato in

tutte le aree dell'organizzazione. In questo caso si parla di **OLTP (Online transaction processing)** e serve per supportare operazioni semplici e quotidiane, e coinvolge "pochi" dati;



- **Sistemi informatici direzionali (DSS)** sono strumenti di supporto ai processi di controllo delle prestazioni aziendali e di decisione manageriale. I dati sono organizzati in Data Warehouse e gestiti da un opportuno sistema che potrebbe essere simile a un DBMS. Le applicazioni che girano sui Data Warehouse vengono chiamate applicazioni **OLAP** e servono a gestire le analisi dei dati come dati storici. Tipicamente presenta operazioni complesse e non ripetitive, e ogni operazione può coinvolgere molti dati. A differenza della base di dati che contiene l'insieme di tutti i dati, in un Data Warehouse si può estrarre un sottoinsieme di dati perché l'obiettivo è studiare un fenomeno. Di solito si ha disposizione una sola base di dati mentre di Data Warehouse se ne possono avere molti.



Differenze tra OLTP e OLAP:

	OLTP	OLAP
Scopi	modificare base di dati	analizzare i dati
Utenti	molti, utenti alla piramide dell'organizzazione, ruolo esecutivo	pochi, dirigenti e analisti
Dati	completi, analitici	sintetici
Usi	noti a priori	poco prevedibili
Quantità di dati per attività	bassa (decine)	alta (milioni)
Aggiornamenti	frequenti	rari
Visione dei dati	corrente	storica
Ottimizzati per	transazioni	analisi dei dati

1.3 Big Data

Si parla di **Big Data** quando i sistemi tradizionali di base di dati e Data Warehouse risultano non adatti per motivi legati al *volume*, *varietà*, *velocità*. Ad esempio una applicazione può appartenere alla categoria Big Data perchè la quantità di dati da analizzare è talmente grande da non poter essere gestita con un DBMS. Applicazioni che generano questa quantità di dati sono tipicamente applicazioni che producono dati in maniera automatica. La fase di "pulizia" dei dati tipica delle base di dati o Data Warehouse non è possibile, allora l'unico approccio è quello del *Data Lake* cioè non fare nessuna operazione di pre-elaborazione. Altri aspetti che mettono in crisi le base di dati o le Data Warehouse sono quando i dati hanno una estrema varietà o vi sono problemi sulla velocità, cioè quando si devono effettuare delle interrogazioni sui dati che hanno un volume tale per cui anche solo scandire i dati è infattibile o per completare progetti in tempi rapidi con sistemi con la complessità di base di dati o Data Warehouse.

1.4 DBMS

Un DBMS è un sistema centralizzato o distribuito che offre opportuni linguaggi per:

- definire lo schema di una base di dati (schema logico);
- in via opzionale si può scegliere le strutture dati per la memorizzazione dei dati (schema fisico);
- inserire i dati rispettando i vincoli definiti nello schema;
- recuperare e modificare i dati interattivamente o da programmi.

Una base di dati è una raccolta di dati permanenti suddivisi in *metadati* e *dati*. Prima vengono inseriti i metadati e soltanto dopo possono essere inseriti i dati. Il DMBS gestisce entrambi: per metadati si indica fatti sulla schema dei dati, cioè "dati che descrivono dati" e sono interrogabili con le stesse modalità previste per i dati. I dati sono rappresentazioni di certi fatti e hanno le seguenti caratteristiche:

- sono organizzati in insiemi strutturati e omogenei correlati fra di loro;
- sono molti, in assoluto e rispetto ai metadati, e non possono essere gestiti in memoria temporanea;
- sono accessibili mediante *transazioni*, unità di lavoro atomiche che non possono avere effetti parziali;
- sono protetti sia da accesso da parte di utenti non autorizzati, sia da corruzione dovuta a malfunzionamento hardware;
- sono utilizzabili contemporaneamente da utenti diversi.

Se manca una di queste caratteristiche, lo strumento utilizzato non è un DBMS. Il modello *relazionale* è il più diffuso fra i DBMS commerciali, dove per relazionale si riferisce alla struttura delle collezioni. Lo schema di una relazione ne definisce nome e descrive la struttura dei possibili elementi della relazione. Per operare, ci si deve connettere al DBMS, si crea una base di dati:

```
CREATE database EsempioEsami                                    -- definizione base di dati
```

ora si possono inserire i metadati, cioè la definizione dello schema dei dati che andranno inseriti:

```
CREATE TABLE Esami (Materia CHAR(5), Candidato CHAR(8),          -- definizione schema
Voto INT, Lode CHAR(1), Data CHAR(6))
```

a questo punto si possono inserire i dati

```
INSERT INTO Esami VALUES ('BDSI1', '080709', 30, 'S', 070900)    -- inserimento dati
```

e interrogare i dati:

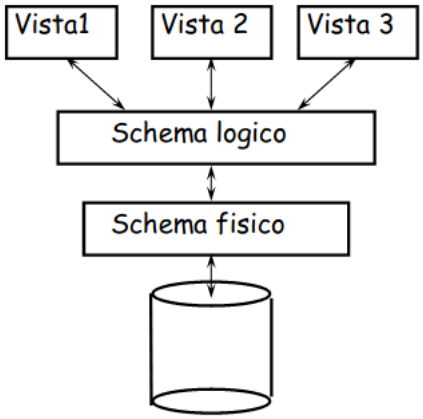
```
SELECT Candidato                                                 -- interrogazione dati
FROM Esami
WHERE Materia = "BDSI1" AND Voto = 30
```

L'interrogazione dei dati è una operazione dichiarativa e il cui algoritmo è deciso dal DBMS.

1.4.1 Funzionalità DBMS

Le funzionalità del DBMS si possono elencare in questo ordine:

- **Linguaggio per la definizione della base di dati**, un DBMS fornisce 3 diversi livelli di descrizione dei dati:
  - *Livello logico*: descrive la struttura degli insiemi di dati e delle relazioni fra loro;
  - *Livello fisico*: descrive come vanno organizzati fisicamente i dati nelle memorie permanenti e quali strutture dati ausiliarie prevedere per facilitarne l'uso. Per esempio decide se la tabella degli studenti viene memorizzata ordinata per cognome, nome o anno di nascita. Questa è una decisione indipendente dal livello logico, e si può cambiare in qualsiasi momento. Le informazioni di livello fisico sono 3: *il modo in cui organizzata ogni singola tabella*, cioè se è disordinata, ordinata rispetto a un livello di un attributo o se è memorizzata secondo una tabella hash oppure albero di ricerca; *quali indici mettere su ciascuna tabella*, cioè una struttura dati che permette di trovare facilmente un particolare dato; *struttura di distribuzione* cioè se vi sono tante macchine, si decide su quali macchine si memorizza una certa tabella un certo numero di volte;



  - *Livello vista logica*: descrive come deve apparire la struttura della base di dati ad una certa applicazione, cioè vengono implementate delle viste che permettono di vedere soltanto un sottoinsieme dello schema logico. Questo facilità l'uso, permette un certo livello di privacy, ma anche un controllo per gli errori, cioè se ci sono dei dati sbagliati nello schema logico, non serve controllare tutto lo schema ma solo quelle applicazioni che si aveva diritto ad accedere a una porzione di schema logico.

  Il fatto di avere 3 livelli ha dei vantaggi importanti, l'*indipendenza fisica*: i programmi non devono essere modificati in seguito a modifiche dell'organizzazione fisica dei dati, cioè non dipendono dallo schema fisico; l'*indipendenza logica*: i programmi non devono essere modificati in seguito a modifiche dello schema logico. Quindi vi è la possibilità di modificare schema fisico e logico senza dover ricompilare il codice sorgente o le applicazioni. I sistemi non accedono allo schema logico, ma soltanto alle viste, quindi quando viene modificato lo schema logico, se è necessario si può cambiare le funzioni delle viste logiche in modo che rimangono uguali, con il vecchio sottosistema di schema logico;

- **Linguaggio per l'uso dei dati**, i DBMS mettono a disposizione un linguaggio di programmazione che permette di creare database, schemi e dati usando un'interfaccia testuale. I DBMS moderni offrono un'interfaccia grafica per accedere ai dati, un linguaggio di interrogazione per gli utenti non programmatori, un linguaggio di programmazione in grado di interagire con il DBMS e un linguaggio per lo sviluppo di interfacce per le applicazioni;

- **Meccanismo per il controllo dei dati**, cioè meccanismi che permettono di definire vincoli di *integrità*, di *sicurezza* da usi non autorizzati e *affidabilità* ovvero protezione dei dati da malfunzionamenti hardware e software, secondo il principio transazionale. Una **transazione** è una sequenza di azioni di lettura e scrittura in memoria permanente e di elaborazioni di dati in memoria temporanea, con le seguenti proprietà:
  - **Atomicità**: le transazioni che terminano prematuramente sono trattate dal sistema come se non fossero mai iniziate, pertanto eventuali loro effetti sulla base di dati sono annullati;
  - **Persistenza**: le modifiche sulla base di dati di una transizione terminata normalmente sono permanenti, cioè non sono alterabili da eventuali malfunzionamenti;
  - **Serializzabilità**: cioè se vi sono una serie di utenti che accedono contemporaneamente agli stessi dati, l'esecuzione delle loro transazioni non crea interferenza, ma l'effetto complessivo è quello di una esecuzione seriale.

Lo svantaggio principale di un DBMS è il fatto che prima di caricare i dati è necessario definire uno schema, e ci sono dei casi in cui avere uno schema è impossibile perchè i dati non obbediscono a nessun schema oppure perchè il tempo di sviluppo è breve e non vi è il tempo di costruire un buon schema. I DBMS possono essere costosi, complessi da installare, mantenere in esercizio e solitamente sono ottimizzati per le applicazioni OLTP ma non per quelle OLAP.

Chapter 2

Modellazione e progettazione

2.1 Progettazione di basi di dati

Progettare una basi di dati vuol dire progettare lo schema e le applicazioni. La progettazione dei dati è l'attività più importante perchè se si sbaglia a implementare una applicazione si deve solamente sistemare la suddetta applicazione, mentre se si sbaglia a progettare i dati si rischia di dover riprogettare tutte le applicazioni che si basano su quei dati. Attorno alla base di dati si muovono 4 ruoli:

- **Committente** è un dirigente che si rende conto che la sua organizzazione ha bisogno di un miglior sistema informatico efficiente, robusto, per gestire i dati;
- **Progettista o Analista** è un dipendente dell'azienda di consulenza informatica. Il progettista interroga il committente e da questo produce un progetto concettuale che dovrà essere approvato dal cliente. Il progetto concettuale è un modello astratto ed è il risultato di un processo di interpretazione che rappresenta formalmente idee e conoscenze di un fenomeno. La rappresentazione è data con un linguaggio formale;
- **Sviluppatori** producono la base di dati e le applicazioni che rispondono al progetto concettuale. A questo punto il codice viene installato dal committente;
- **Data Base Admnistrator (DBA)** gestisce utenti e sistema.

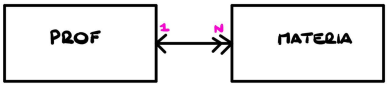
La progettazione delle basi di dati avviene in 4 fasi distinte e ciascuna fase è centrata sulla modellazione: **analisi dei requisiti**: si intervista il committente per sapere di cosa ha bisogno e questa fase termina con la **progettazione concettuale**. Una volta che il committente approva il progetto, i programmatori lo traducono in un **progetto logico** cioè in un linguaggio di programmazione. Infine dopo la progettazione logica ci sarà la **progettazione fisica**.

2.2 Modellazione

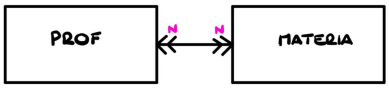
Gli aspetti del problema di modellazione sono l'**aspetto ontologico** cioè la conoscenza concreta dei fatti di come è fatto il mondo ed è diviso in *conoscenza astratta* ovvero la struttura informativa e vincoli sulla conoscenza concreta, e *procedurale* ovvero le operazioni che esistono per operare sui dati. L'**aspetto logico** cioè è un insieme di meccanismi di astrazione per descrivere la struttura della conoscenza concreta. I fatti che si vogliono rappresentare sono: le *entità* con le loro proprietà, le *collezioni* di entità omogenee e le *associazioni* fra entità. Le classi di uno schema si distinguono in tre categorie:

- *oggetti fisici*: gli elementi hanno proprietà fisiche e quindi tipicamente occupano una posizione nello spazio;
- *eventi*: si sviluppano nel tempo e hanno una localizzazione temporale, cioè sono avvenimenti che succedono in qualche momento;
- *classi astratte*: generalmente sono modelli per qualche classe concreta.

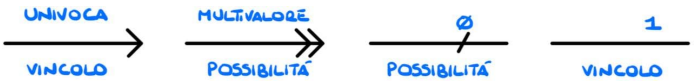
Un'associazione si divide in *uno a uno, uno a molti, molti a molti*. Per esempio in una scuola dove ogni materia è tenuta da un solo professore, l'associazione sarà:



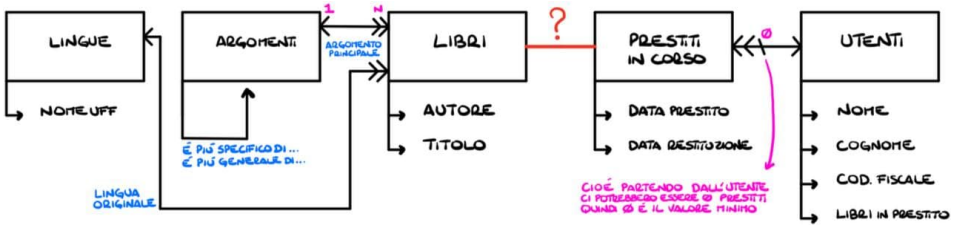
cioè significa che associa molte materie ad un professore e nell'altra direzione associa un insegnante a una materia. Quindi in un certo senso va da 1 a N in una direzione, e da 1 a 1 nell'altra direzione. Mentre se in una scuola ogni materia può avere più insegnanti allora:



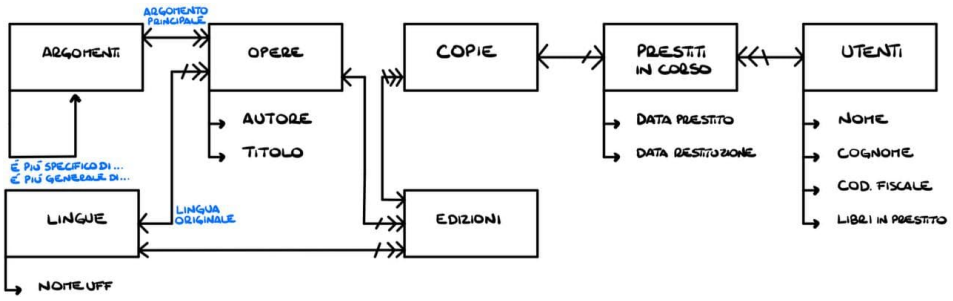
ovvero associa un professore a più materie, e una materia a più professori. Un'associazione può essere univoca o multivalore. Un'associazione univoca è un vincolo, perché per esempio in una scuola "per ogni materia ci *deve* essere un insegnante", mentre un'associazione multivalore è una possibilità in più "per ogni materia ci *possono* essere più insegnanti" ma se per un periodo ogni materia è insegnata da un unico insegnante è corretto usare l'associazione multivalore. Per quanto riguarda l'opzionalità ci può essere come minimo zero o uno. L'opzione con uno è un vincolo perchè non può esistere una materia senza insegnante, mentre l'opzione zero indica che una materia non deve necessariamente avere un'insegnante. Inoltre il caso zero permette anche il caso uno, cioè permette di avere l'insegnante e in più permette anche di non averlo.



Esercizio 1. Si studia il caso di dover gestire una biblioteca in cui si possono prendere in prestito dei libri e per ogni libro interessa conoscere autore, titolo, lingua e argomento principale. Per ogni lingua interessa raccogliere il nome ufficiale. Interessano anche gli argomenti trattati nei libri che sono definiti a priori ed esiste una relazione di generalizzazione, cioè uno è più generale di un altro. I libri possono essere prestati e per ogni prestito in corso interessa la data del prestito, la data di restituzione, il libro e l'utente che lo ha preso in prestito. Per ogni utente interessano nome, cognome, codice fiscale e libri attualmente in prestito.

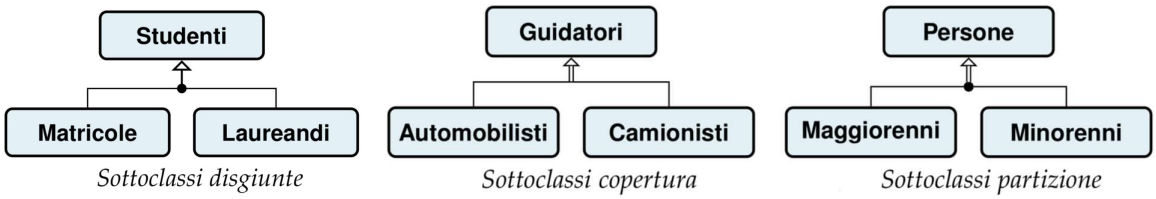


Questa è la realtà definita dal committente, da essa il DBA visualizza le classi con degli attributi. Ogni *argomento* può essere più generale o più specifico di altri argomenti, questa informazione viene indicata con una associazione. Un'associazione è un insieme di frasi binarie omogenee che hanno il soggetto in una certa classe e il complemento in un'altra, ad esempio "la lingua originale della divina commedia è l'italiano", "la lingua originale di aspettando Godot è il francese", "la lingua originale dei racconti di Canterbury è l'inglese", tutte queste frasi binarie hanno tutte lo stesso verbo: *la lingua originale è*, hanno tutte il soggetto nella classe delle lingue e il complemento nella classe delle opere. A questo punto, tornando alla questione degli argomenti: il soggetto è un argomento e il complemento è un argomento. Per rendere più chiara una associazione, a volte si aggiunge una frase. Dato un utente si possono trovare molti prestiti e come minimo nessuno. In generale tutte le volte che una associazione è multivalore, si ammette quasi sempre il caso zero, ma con delle eccezioni: un prestito è associato al minimo ad una persona, e non zero perchè il prestito è l'evento in cui si presta un libro ad un utente, per cui senza utente il prestito non esisterebbe. L'associazione *libri - prestiti in corso* non può essere decisa dal DBA, ma dipende da come il committente lavora. Si suppone che per il cliente un prestito equivale a uno solo libro, più difficile la direzione opposta, cioè *dato un libro quanti prestiti in corso sono possibili contemporaneamente?* Si deve disambiguare la questione dei *libri*: da una parte ci sono i libri identificati da titolo e autore che d'ora in poi saranno chiamati *opere* e dall'altra ci sono le *copie* del libro. A questo punto si devono dividere gli attributi dei libri tra *opere* e *copie*. Vi è il problema della lingua perché una opera è stata scritta in una lingua, ma può essere tradotta in altre lingue, per risolvere questa difficoltà si può aggiungere l'entità *edizioni*.



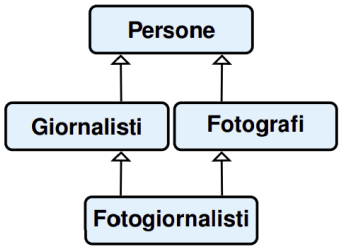
2.2.1 Sottoclassi

Spesso all'interno delle collezioni ci sono delle *sottoclassi* che servono per ottenere informazioni extra. Gli elementi di una sottoclasse sono un sottoinsieme degli elementi della superclasse per cui vi è una relazione di ereditarietà dell'informazione: la sottoclasse degli *studenti* hanno tutte le informazioni delle *persone*. Quando vengono definiti due diverse sottoclassi di una stessa superclasse, possono essere correlate da un vincolo di *disgiunzione*, cioè è impossibile appartenere a entrambi le sottoclassi, di *copertura*, cioè l'unione delle sottoclassi copre interamente la superclasse però non vi è disgiunzione perchè è possibile appartenere a entrambi le sottoclassi, o di *partizionamento*, cioè copre interamente la sottoclasse ma vi è disgiunzione.

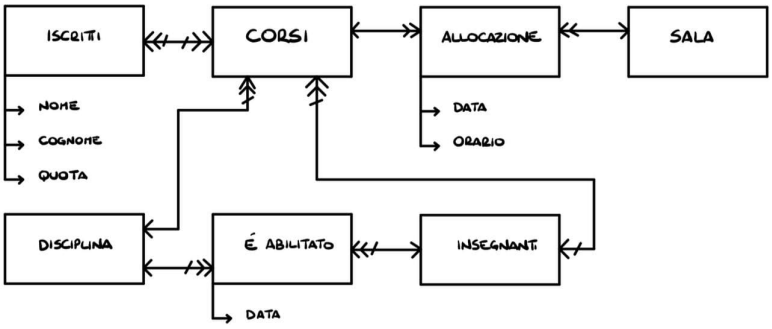


Sottoclassi disgiunte *Sottoclassi copertura* *Sottoclassi partizione*

Nel modello a oggetti vi è una distinzione tra i tipi degli oggetti, cioè definisce la struttura degli oggetti, e la classe cioè una collezione che serve per contenere oggetti dello stesso tipo. Esistono anche *gerarchie multiple*, cioè non è un albero ma è tipicamente un grafo diretto aciclico.

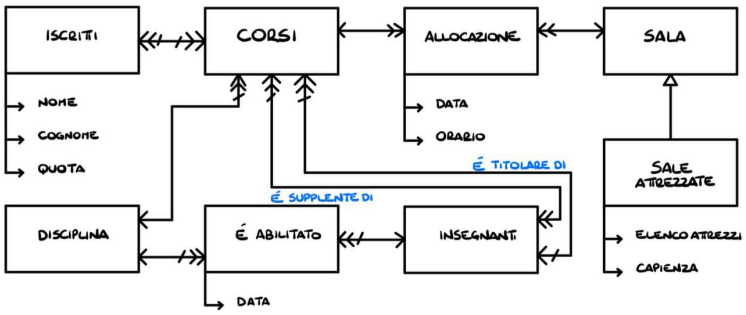


Esercizio 2. Si analizza il caso di gestione di una palestra in cui vengono tenuti dei corsi da degli insegnanti. Gli istruttori sono abilitati a insegnare corsi di determinate discipline. Interessa sapere anche in quale data un istruttore è stato abilitato. Ogni corso ha un solo insegnante e si tengono in certe sale in certi orari. Ogni corso si può svolgere in più sale e in una sala si possono svolgere più corsi o lasciarla vuota per un periodo. Per gli iscritti alla palestra interessa tenere traccia oltre alle informazioni anagrafiche, anche delle quote di iscrizione e dei corsi a cui sono iscritti.



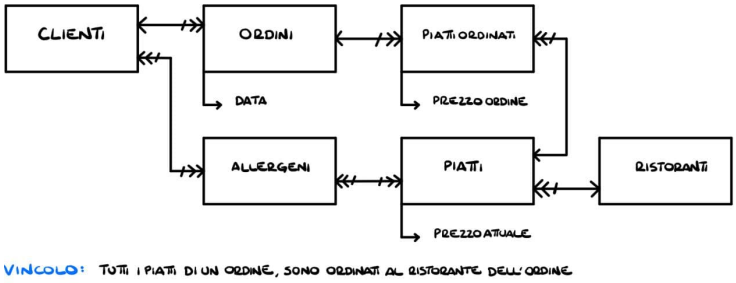
Si può immaginare che un corso esiste solo nel momento in cui c'è l'istruttore, prima di esso si limita ad essere una disciplina, quindi dato un corso ci sono zero o un'insegnante. Data una disciplina ci possono essere diversi corsi contemporaneamente o nessuna. L'associazione *insegnamenti - corsi - discipline* permette di sapere se l'insegnante è abilitato a svolgere una determinata disciplina, ma dato un istruttore abilitato non necessariamente insegna una disciplina perché ci potrebbe essere un insegnante che sarebbe abilitato a fare aerobica e kick box ma in questo momento sta insegnando solamente aerobica quindi occorre anche l'associazione molti a molti *discipline - insegnanti*. Le date sono sempre attributi perché non interessa collezionare periodi, ma eventi che sono accaduti in una certa data. La data di abilitazione dell'insegnante non è un attributo dell'istruttore perché se è abilitato in tre discipline, l'abilitazione avrà tre date diverse, quindi sarà un attributo dell'associazione *discipline - insegnanti*. Un'associazione con attributi è una collezione di frasi ternarie: "L'insegnante Mario si è abilitato alla disciplina aerobica in data 1 marzo 1991" e si indica con una linea tratteggiata. Molto spesso le associazioni con attributi si incontrano con i collegamenti molti a molti, ed è consigliabile *promuovere* l'associazione ad entità con associazioni uno a uno. Non è consigliabile promuovere una associazione a classe quando non si riesce a trovare un nome per quella associazione, cioè non si inventa dei nomi insensati per promuovere l'associazione a classe. Per quanto riguarda gli orari se un

certo corso si tiene il lunedì e il mercoledì alle 9 nella sala 1 e 3, non si può aggiungere *orario* come attributo di *corsi* perchè non si sa come collegare per esempio il mercoledì alla sala 1, per cui si può aggiungere orario come attributo della associazione o promuovere l'associazione ad una nuova collezione. Infatti questa è collezione ternaria: "il corso aerobica si tiene nella sala 1 il mercoledì alle 3". La nuova collezione la si chiama *allocazione* con attributi data e orario. L'associazione *iscritti - corsi* è una "normale" associazione molti a molti perchè non interessa sapere in che data un utente si è iscritto a un determinato corso, se ha pagato una quota speciale per un determinato corso o se ha presentato un certificato. Le quote pagate può essere un *attributo multivalore* perchè può contenere più valori, e può essere anche *strutturato* perchè per ogni quota pagata possiamo ricordare la quota, la data del pagamento e il metodo di pagamento.



Se si immagina esiste un sottoinsieme di sale attrezzate per il quale interessa memorizzare gli attrezzi allora ci sarà la sottoclasse *sale attrezzate*. Le sottoclassi vengono realizzate solo se ci sono degli attributi in più, normalmente più di uno, per esempio elenco attrezzi e capienza che possono essere vuoti o usati ma contemporaneamente. Si immagina ancora che ogni corso oltre ad avere un insegnante titolare ha un insieme di supplenti. Per rappresentarlo non si utilizza una sottoclasse perchè per esempio un istruttore può essere il titolare di aerobica e supplente di kick box, quindi non ha a che vedere con l'insegnante ma è una questione relazionale tra *corsi - insegnanti*: si aggiunge una seconda associazione. Una associazione è un insieme di frasi che hanno lo stesso verbo ma in questo caso ce ne sono due. Un corso può avere molti supplenti, e un insegnante può essere supplente di molti corsi.

Esercizio 3. Un'azienda che offre la consegna di cibo a domicilio è interessata a mantenere informazioni sui ristoranti associati e sui clienti. Per un cliente interessano nome, indirizzo, età, e la lista di tutti gli ordini effettuati nel passato. Ogni ordine è stato effettuato da un cliente, in una certa data, e interessano anche il ristorante a cui era diretto e la lista dei piatti ordinati. Per ogni piatto interessa il nome, la foto, il ristorante che lo offre (che è unico), ed il prezzo a cui è attualmente offerto – uno stesso piatto appare ovviamente in molti ordini diversi. Quando si memorizza un ordine, per ciascun piatto dell'ordine interessa il prezzo a cui è stato venduto, dato che tale prezzo potrebbe non coincidere con il prezzo attuale. Tutti i piatti di un ordine sono offerti dal ristorante a cui è diretto. L'azienda mantiene anche una lista di allergeni, per ciascuno dei quali memorizza il nome e le patologie per le quali è controindicato. Per ogni piatto si mantiene la lista di tutti gli allergeni presenti. Lo stesso allergene può apparire in più piatti. Per ogni clienti l'azienda mantiene la lista degli allergeni ai quali si è dichiarato intollerante.



Un ordine è associato a un cliente, e dato un cliente sono associati nessuno o molti ordini. Ogni piatto è realizzato da un unico ristorante, cioè la carbonara del ristorante Roma o del ristorante Luigi sono due piatti differenti. *Prezzi* non è un'entità, ma un attributo di *piatti* perchè non interessa raccogliere informazioni su €17.50, che ha un qualche interesse solo nel momento in cui esiste un piatto con quel prezzo. Quindi in *piatti* ci sarà l'attributo *prezzo attuale*, mentre l'associazione molti a molti *ordini - piatti* va l'attributo *prezzo ordine*, che viene promossa nell'entità *piatti ordinati*. L'associazione *ordini - ristoranti* permette di ordinare senza nessun piatto, se invece si obbliga di avere in ciascun ordine almeno un piatto allora l'associazione diventa ridondante. Si considera il caso in cui l'associazione *ordini - ristorante* è ridondante e quindi non viene inserita, in questo contesto si ha bisogno di un vincolo: *tutti i piatti di un ordine, sono ordinati al ristorante dell'ordine*, senza questo vincolo si potrebbe ordinare dal ristorante Roma, l'amatriciana del ristorante Luigi. La lista delle patologie a cui un allergene è associato può essere intesa come una collezione o attributo multivalore, questa è una questione da chiedere al committente.

Chapter 3

Modello relazionale

3.1 Definizione

Il progettista traduce il modello concettuale deciso con il committente in un modello logico usando il formalismo relazionale. Nel modello relazionale anzichè avere delle classi, che sono delle rappresentazioni astratte della realtà, si utilizzano delle relazioni chiamate anche tabelle che sono delle collezioni di record. Ogni relazione possiede un nome, uno schema che descrive la struttura della relazione e infine una estensione che è un insieme di righe che rappresentano i dati. Lo schema sono i metadati mentre le righe sono i dati. Lo schema viene anche chiamato *schema di relazione*, mentre i dati *istanza di relazione*. Tutte queste diverse relazioni possono essere connesse fra di loro con un meccanismo di **chiavi primarie e esterne**.

Studenti
↑
NOME

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

← SCHEMA METADATI

DATI
ISTANZA DI RELAZIONE

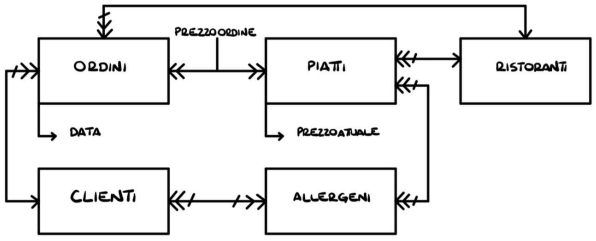
Il modello relazione è definito a partire dal concetto di **ennubla** e relazioni. Un tipo ennubla è un insieme finito di coppie, dove ogni coppia è definito come *attributo - tipo elementare*. Se T è un tipo ennupla, R(T) è lo schema della relazione R. Lo schema di una base di dati è un insieme di schemi di relazione Ri(Ti). Un'istanza di uno schema R(T) è un insieme finito di ennuple di tipo T. Due tipi ennupla sono uguali quando hanno lo stesso insieme coppie *attributo - tipo*. Un concetto fondamentale nei sistemi relazionali è il concetto di **chiave**:

- **Superchiave:** è un insieme di attributi che identifica univocamente le righe della tabella. Nella figura, la coppia *nome - matricola* è superchiave perchè non è possibile avere due righe diverse che hanno lo stesso nome e la stessa matricola. Quando due righe hanno lo stesso nome e la stessa matricola sono uguali. Anche la colonna *matricola* è superchiave perchè non si può avere due righe diverse che hanno la stessa matricola;
- **Chiave:** è una superchiave minimale nel senso insiemistico. Nell'esempio, matricola è sia superchiave perchè identifica la riga, che chiave perchè se si cancella un qualunque attributo non è più chiave dato che si ottiene l'insieme vuoto. La coppia *nome - matricola* è superchiave perchè identifica la riga ma non è chiave perchè l'attributo nome può essere cancellato;
- **Chiave primaria:** è una chiave che il progettista della base di dati ha scelto come più adatto per identificare l'entità. Una singola tabella può avere tante chiavi, se si aggiungesse a studenti gli attributi *codice fiscale e ID* allora ci sarebbero tre chiavi, cioè matricola, codice fiscale e ID. Quando il progettista, crea la tabella studenti, comunica al DBMS quale delle tre chiavi è primaria. La chiave primaria deve essere **immutabile** nel tempo, è buona norma usare un attributo, chiamata **chiave primaria artificiale**, non noto al committente e assegnare un numero arbitrario.
- **Chiave esterna:** è una associazione tra un attributo di una tabella A e una tabella B. La chiave esterna è un vincolo di sottoinsieme che indica che i valori assunti dalla colonna chiave esterna sono un sottoinsieme dei valori assunti dalla chiave primaria della tabella a cui fa riferimento. Ogni elemento di A appartiene a B, ma non è vero il viceversa. Una chiave esterna è un attributo che può essere usata per rappresentare una associazione. Non fa confusa con le associazioni che si trovano nel modello a oggetti.

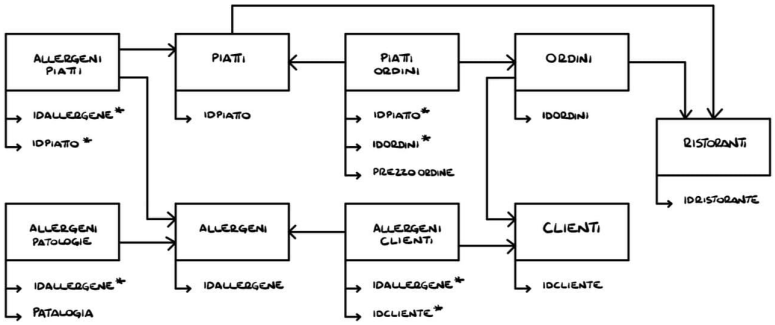
La chiave primaria si rappresenta con una sottolineatura, mentre la chiave esterna oltre la sottolineatura si aggiunge un asterisco.

3.2 Trasformazione di schemi a oggetti in relazionali

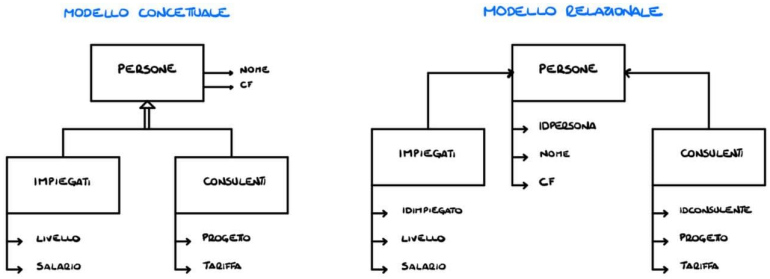
Si immagina di tradurre lo schema ad oggetti del ristorante, con una soluzione diversa, in modello relazionale.



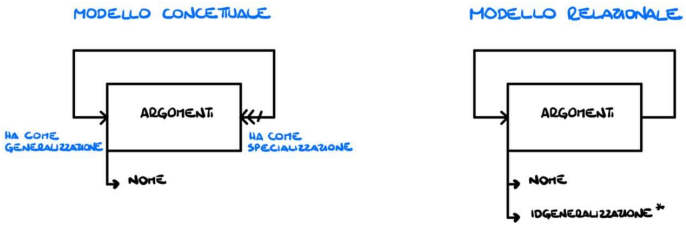
Per ogni classe si costruisce una tabella corrispondente e per ogni tabella si sceglie un attributo che sarà la chiave primaria, solitamente una chiave primaria artificiale come IDRistorante o IDCliente, e a questo attributo nuovo, si aggiungono tutti gli attributi che c'erano prima. Ora si traducono le associazioni: le associazioni univoche diventano chiavi quindi, per esempio, quella tra *piatti - ristoranti* parte da piatti e arriva a ristoranti. Nella tabella da dove parte la freccia ci sarà la chiave esterna della tabella di arrivo. Le associazioni molti a molti si crea una **tabella di coppie** *allergeniClienti* che contiene solo due attributi: IDAllergene e IDCliente, questa tecnica viene usata anche con associazioni con attributi. Per le associazioni uno a uno, nel primo esempio della biblioteca, si può risolvere mettendo all'interno di copie l'identificativo del prestito in corso, o dentro il prestito l'identificativo del libro a cui quel prestito si riferisce. Per gli attributi multivalore come per esempio *patologie* lo si inserisce in una tabella di coppie *allergeniPatalogie* con l'attributo patalogie.



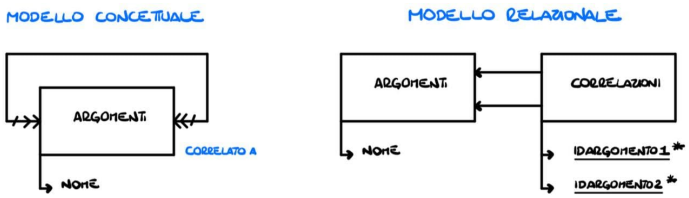
Tutti i DBMS ammettono che un identificativo può rimanere vuoto: **NULL** che ha il significativo *valore non noto* o *valore non applicabile*. Quando un campo è chiave primaria, è implicitamente indicato come *NOT NULL*: un campo chiave primaria non può mai essere nullo, viceversa, se si vuole indicare che una chiave esterna sia un campo NOT NULL va scritto implicitamente. Si immagina di avere una classe persona, con attributi nome e codice fiscale, partizionata in due sottoclassi, impiegati e consulenti, e con gli attributi come in figura. Per tradurlo nel modello relazionale:



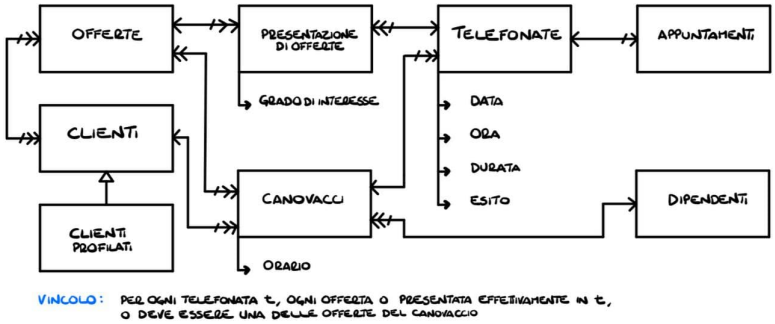
Si studia il caso di avere un tabella argomenti con un associazioni uno a molti "ha come specializzazione" e un'altra uno a uno con "ha come generalizzazione" e il relativo modello relazionale:



In un'altra tabella argomenti con associazione molti a molti "correlato a" nel modello relazionale si rappresenta con una tabella di correlazioni con due campi argomento 1 e argomento 2 che sono sia chiavi primarie che secondari, e come tutte le chiavi primarie sono NOT NULL:

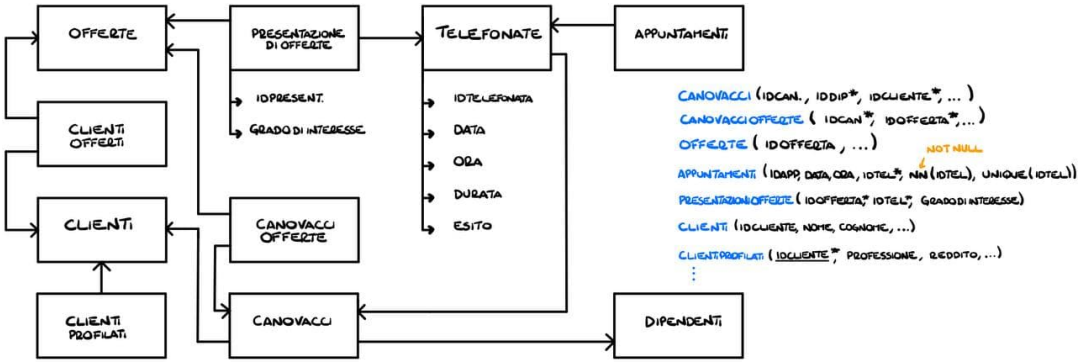


Esercizio 4. Una società di telecomunicazioni effettua campagne telefoniche per proporre offerte ai propri clienti, ed è interessata a gestire i dati di queste telefonate. Per ogni telefonata viene preparato un canovaccio, che specifica il dipendente che chiamerà, il cliente destinatario, l'orario indicativo per il primo tentativo e le offerte che saranno presentate. Per ogni canovaccio che gli viene assegnato, il dipendente effettua una o più telefonate, fino a che non è riuscito a parlare effettivamente con in cliente e a concludere, in qualche modo, la presentazione del canovaccio.Per ogni telefonata effettuata interessa memorizzare data, ora, durata, esito (intermedia o finale), il canovaccio a cui si riferisce, e le offerte che sono state effettivamente presentate durante la telefonata, dato che in genere non risulta possibile presentare tutte le offerte che erano state specificate nel canovaccio. Inoltre, per ciascuna offerta presentata in una specifica telefonata, interessa memorizzare il grado di interesse (su una scala da -5 a +5) evidenziato dal cliente per tale offerta. Infine, se durante la telefonata il dipendente è riuscito a fissare un appuntamento con il cliente, interessa anche memorizzare i dati relativi all'appuntamento, ed il particolare la telefonata durante la quale è stato fissato, l'ora e il luogo dell'appuntamento, ed eventuali annotazioni. Per ogni cliente interessano, oltre a tutti gli appuntamenti che lo riguardano, anche nome, cognome, indirizzo e le offerte che ha sottoscritto nel passato. Per un sottoinsieme di clienti profilati interessa anche la professione, il reddito stimato, la condizione familiare. Per ogni offerta interessa una descrizione testuale ed un intervallo di validità.Per ogni dipendente, interessano nome, cognome, indirizzo e qualifica.



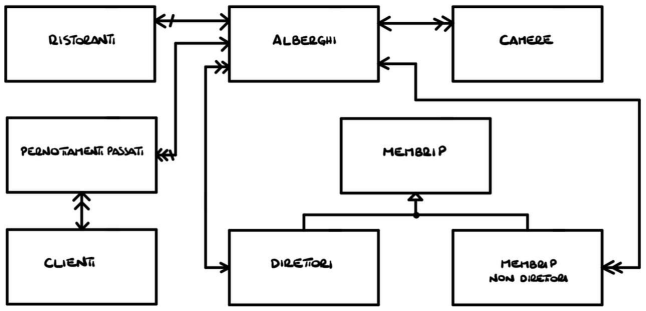
Per un singolo canovaccio il dipendente può effettuare più telefonate, per ogni telefonata il dipendente segue un singolo canovaccio. A un dipendente gli saranno assegnati molti canovacci, e il singolo canovaccio è assegnato a un solo dipendente. Un cliente può essere associato a più canovacci, un canovaccio è associato a un singolo cliente. Un canovaccio comprende diverse offerte, una offerta sarà presentata in più canovacci. Interessa sapere quali offerte sono state effettivamente presentate durante una telefonata, quindi la si vede come una ulteriore associazioni *telefonate - offerte* perchè a fronte di un canovaccio si possono effettuare anche due telefonate, perchè per esempio nella prima telefonata è caduta la linea, e quindi si fa un'altra telefonata. Alcune offerte possono essere presentate nella prima telefonata, mentre altre nella seconda, quindi ci sono le offerte che si speravano di presentare (canovaccio), e le offerte effettivamente presentate. Il grado di interesse è un attributo dell'associazione *telefonate - offerte*, quindi l'associazione con attributi viene promossa nell'entità *PresentazioneDiOfferte*. Il committente potrebbe volere il vincolo d'inclusione *per ogni telefonata T e per ogni offerta O presentata effettivamente in T, O deve essere una delle offerte del canovaccio*, cioè che è impossibile che un dipendente propone una offerta al di fuori del canovaccio. Ogni appuntamento è stato fissato durante una telefonata, una telefonata può fissare al massimo un appuntamento. Per ogni cliente interessano le offerte passate, e una offerta può essere associata a più clienti. Infine è presente una sottoclasse *clienti profilati* sui quali vi sono informazioni maggiori.

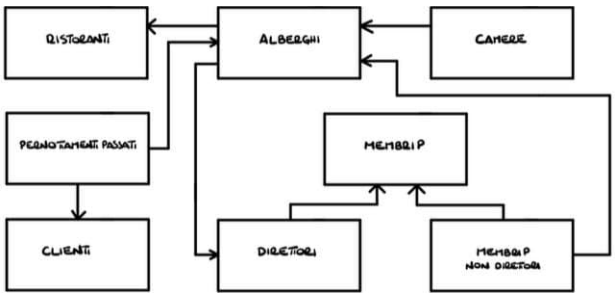
Il relativo modello relazionale:



Nell'associazione *appuntamenti - telefonate*, si riesce a catturare la totalità dell'associazione (appuntamenti → telefonate) con *NOT NULL(IDTel)*, mentre l'univocità (telefonate → appuntamenti) la si cattura con **Unique(IDTel)**, cioè il vincolo Unique indica che non si possono avere due appuntamenti diversi con la stessa telefonata. La chiave della tabella clienti profilati sarà sia chiave primaria che esterna.

Esercizio 5. Una catena alberghiera è costituita da diversi alberghi, identificati da un codice: ogni albergo ha un nome, un indirizzo, una categoria. In ciascun albergo una camera è identificata da un numero e le sue caratteristiche sono costituite dal piano, il numero di letti ed il prezzo. I membri del personale, compreso il direttore, sono identificati da un codice e di essi interessa il cognome, il nome e l'anno di nascita. Del direttore interessa anche il titolo di studio. Il direttore può dirigere più alberghi, mentre gli altri dipendenti prestano la loro opera in un solo albergo. Ogni albergo ha diversi membri del personale ma ha un solo direttore. Un albergo può avere annesso un ristorante, e al più uno; in tal caso è specificato l'orario di apertura, l'eventuale giorno di chiusura, i piatti tipici. Viene anche tenuto un archivio dei clienti, di coloro cioè che sono già stati ospiti degli alberghi: di essi interessa cognome, nome, indirizzo ed il numero totale di notti che hanno trascorso in ciascun albergo (solo il numero di notti, non la data od altro) sono identificati da un codice. Si chiede di rappresentare lo schema concettuale in formato grafico, riportando anche tutti gli attributi, lo schema relazionale in formato grafico (rettangoli e frecce) senza attributi, ed infine lo schema relazionale in formato testuale. I vincoli che non possono essere rappresentati graficamente vanno scritti in lingua naturale sotto lo schema grafico.





MODELLO RELAZIONALE

MembriP(idMP, nome, cognome, ...)
Direttori(idMP*, dataNomina)
MembriPNondir(idMP*, qualifica, stipendio, idAlbergo*)
Albergo(idAlbergo, nome, indirizzo, idDirettore*)
Ristoranti(idRistorante, ..., idAlbergo*)
Camere(idCamera, ..., idAlbergo*)
PernottamentiPassati(IdCliente*, idAlbergo*, nottiPassate)
Clienti(idCleinte, nome, cognome, indirizzo)

3.3 Linguaggi relazionali

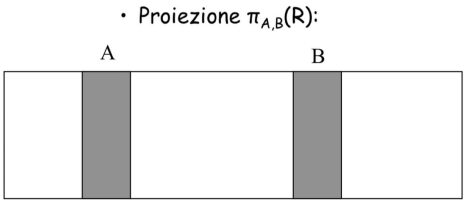
Per linguaggi relazionali si intende come si interroga una base di dati relazionale, esistono due famiglie:

- **Algebra relazionale**: è un insieme di operatori che hanno la caratteristica di ritornare un valore che risiede nello stesso dominio degli argomenti dell'operatore. Sulle struttura matematica della relazione sono state definite un insieme di operatori algebrici che definiscono un linguaggio che è utile per esprimere interrogazioni. L'algebra relazionale assomiglia molto al meccanismo che i DBMS usano come rappresentazione interna delle interrogazioni;
- **Calcolo relazionale**: è un linguaggio dichiarativo di tipo logico dal quale è stato derivato l'**SQL**.

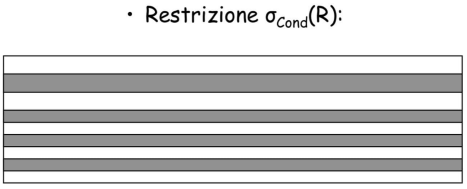
3.3.1 Algebra relazionale

Gli operatori sono:

- **Proiezione**, si immagina di avere una tabella degli studenti con tanti attributi come il nome e l'età. La proiezione essendo una operazione insiemistica elimina tutte le colonne a cui non si è interessati, per esempio tutte le colonne tranne nome e età, ed rimuove i duplicati dal risultato, per esempio se ci sono due studenti che si chiamano Mario e hanno 20 anni, nel risultato ce ne sarà solo uno. La proiezione si indica con il simbolo π e le colonne su cui si va a proiettare;



- **Restrizione**, elimina alcune righe, quindi si indica una condizione *cond* per esempio *età* $\geq$ *25 AND nome* $\neq$ *Mario* nella tabella si manterranno tutte e sole le righe che preservano questo nome. La restrizione si indica con il simbolo σ e la condizione. Il linguaggio a disposizione per esprimere la condizione è dato da:
  - Cond ::= Espr Theta Espr | Cond And Cond | Not Cond
  - Espr ::= Attributo | Costante | Espr Op Espr
  - Theta ::= = | < | > | != | <= | >=
  - Op ::= + | - | * | StringConcat



- **Prodotto**, è applicabile solo tra due tabelle che hanno tutti nomi di attributi diversi. Per esempio R *x* S, dà come risultato una tabella con gli attributi di R + gli attributi di S, dove il + sarebbe l'unione di due insiemi disgiunti. Il prodotto è seguito quasi sempre da una restrizione. L'operazione è fondamentalmente un prodotto cartesiano;

• Prodotto: R × S

a	A
a1	A1
a2	A2

×

b	B
b1	B1
b2	B2
b3	B3

=

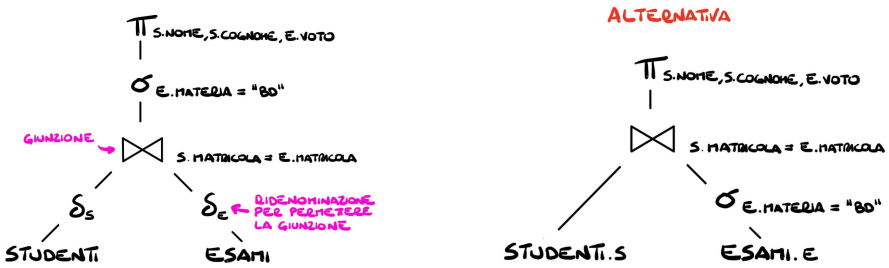
a	A	b	B
a1	A1	b1	B1
a1	A1	b2	B2
a1	A1	b3	B3
a2	A2	b1	B1
a2	A2	b2	B2
a2	A2	b3	B3

- **Ridenominazione**, dato che il prodotto cartesiano ha come necessità di avere tutti nomi diversi, per fare il prodotto di una tabella per sé stessa è necessario ridenominarla. Se si vuole fare il prodotto R *x* R, si prende la tabella R, si ridenomina A con A', B con B' e si moltiplica per la tabella R. La ridenominazione si indica con il simbolo δ;
- **Unione**, R $\cup$ S è l'unione insiemistica ma con il vincolo che è ammissibile solo se il tipo relazione di R è uguale al tipo relazione di S
- **Differenza**, è un operatore insiemistico, cioè R - S sono tutti gli elementi che stanno in R ma non stanno in S. Anche per la differenza è obbligatorio che entrambe le tabelle abbiano lo stesso schema;
- **Intersezione** è un operatore derivato: si suppone ci sia la tabelle R con attributi A e B, e S con attributi A e B. L'intersezione R $\cap$ S si svolge facendo il prodotto tra i due, ma per farlo occorre prima ridenominarli: R x δ_SS. Ora si va a restringere in tutte le righe in cui A è uguale a S.A e B è uguale a S.B cioè in simboli $\sigma_{A = S.A \text{ AND } B = S.B}$. A questo punto si effettua la proiezione su A e B: $\pi_{A,B}$. Mettendo tutto insieme: $\pi_{A,B}(\sigma_{A = S.A \text{ AND } B = S.B}(R \text{ x } \delta_S S))$

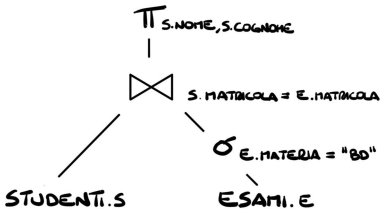
- **Giunzione** si immagina di avere due tabelle: Studenti(matricola, nome, cognome), Esami(matricola, data, voto, materia) e si vuole avere come risultato una tabella EsamiStudenti(matricola, data, voto, nome, cognome). Per generare questa tabella si ridenomina gli studenti δ_S(Studenti), questo permette di fare il prodotto Esami x δ_S(Studenti). Ora si va a restringere alle coppie in cui matricola è uguale a S.matricola. Quindi la giunzione è un prodotto seguito da restrizione.

Esercizio. Partendo dalle tabelle Studenti(matricola, nome, cognome) e Esami(matricola, data, materia, voto) si vuole conoscere:

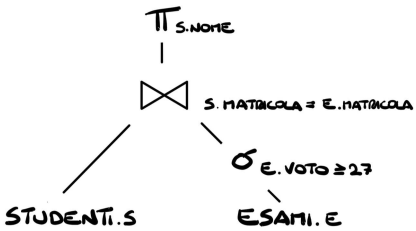
- *Nome, cognome e voto di tutti gli studenti che hanno sostenuto l'esame di BD*. Quando si scrive, per esempio, studenti.s si intende ridenominare tutti i campi di studenti mettendoci una s davanti. La versione alternativa è utilizzare la restrizione prima della giunzione. Dal punto di vista algebrico questa operazione si può sempre fare, cioè quando una restrizione riguarda solo attributi che vengono da una sola tabella;



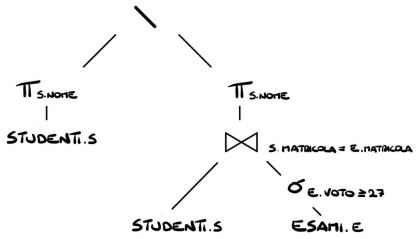
- *Nome, cognome di tutti gli studenti che hanno hanno sostenuto l'esame di BD*. Basta togliere E.voto dalla proiezione finale. Nel momento in cui interessa l'input che arriva solamente da sinistra e non interessa nessun attributo del lato destro (o viceversa) significa che la giunzione serve per implementare un quantificatore esistenziale, cioè *se esiste un esame legato allo studente*. Quindi la giunzione ha un doppio effetto: di connessione e di filtraggio;



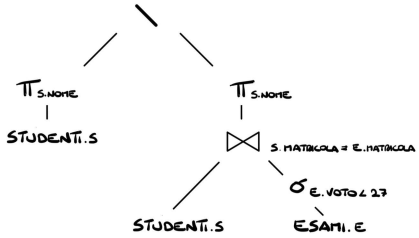
- $\{S.nome \mid S \in Studenti\ .\ \exists\ E \in Esami\text{: } E.matricola = S.matricola \wedge E.voto \geq 27\}$;



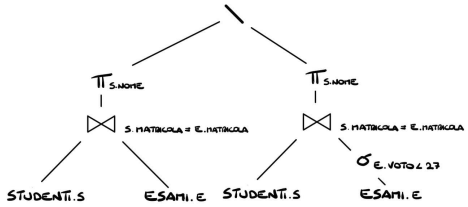
- $\{S.nome \mid S \in Studenti\ .\ \neg\ \exists\ E \in Esami\text{: } E.matricola = S.matricola \wedge E.voto \geq 27\}$, cioè tutti gli studenti meno quelli che hanno superato almeno un esame con voto maggiore a 27. Qui è utile l'operatore di sottrazione: occorre la proiezione sul nome perchè per fare la differenza servono gli stessi campi. Va scritto necessariamente in questo modo, cioè a sinistra il sottraendo e a destra il sottrattore;



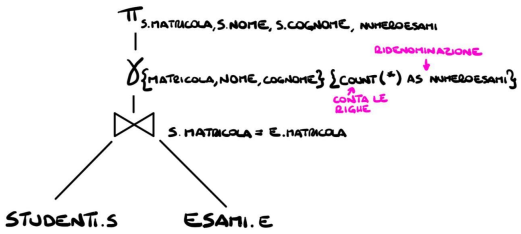
- $\{S.nome \mid S \in Studenti\ .\ \forall\ E \in Esami: E.matricola = S.matricola \Rightarrow E.voto \geq 27\}$, cioè il nome di tutti gli studenti tale che per ogni esame che hanno superato il voto è maggiore di 27. Questa query corrisponde al nome di tutti gli studenti tale che non esiste E in esami, dove E.matricola è uguale a S.matricola e il voto è minore di 27 .



- *Nome di tutti gli studenti che hanno preso solo voti superiori al 27 e che hanno preso almeno un voto superiore al 27*, cioè gli studenti che hanno superato almeno un esame meno quelli che non hanno mai preso voti inferiori al 27;



Un altro operatore, che non fa parte dell'algebra relazionale di base, è il **raggruppamento**, che si indica con il simbolo γ. Si immagina di avere le due tabelle Studenti e Esami, e di voler contare per ogni studente nome, cognome e il numero di esami sostenuti dallo studente. La group by raggruppa per nome e cognome ma nel caso di omonimi serve anche la matricola. In generale è buona norma proiettare tutti gli attributi raggruppati. Per rendere più "elegante" si può ridenominare count(*) direttamente all'interno del group by.



Chapter 4

SQL: per l'uso interrativo di basi di dati

4.1 Struttura

L'algebra relazione è un linguaggio che viene utilizzato internamente dal data base per tradurre le interrogazioni che vengono scritte dal programmatore. Lo sviluppatore, per interrogare il data base, utilizza un linguaggio che fa parte della famiglia dei calcoli relazionali, cioè dei linguaggi che prendono il loro potere espressivo dalla capacità di esprimere quantificatori e prende il nome di SQL. Il Structured Query Language è stato definito nel 1973 ed è oggi il linguaggio universale dei sistemi relazionali, è comprende tre sottolinguaggi:

- **Data Definition Language (DDL)**, è quel sottoinsieme dell'SQL che permette di definire i metadati e la struttura dei dati, cioè comprende comandi come *create table, create data base, ...*
- **Data Manipulation Language (DML)**, è quella parte del linguaggio che contiene comandi per manipolare i dati;
- **Query language**, quella parte del linguaggio che permette di scrivere interrogazioni e corrisponde come potere espressivo all'algebra relazionale.

SQL è un calcolo su multiinsiemi, cioè non opera automaticamente la cancellazione dei duplicati. Si può tornare da multiinsiemi a insiemi con l'operatore **DISTINCT** che elimina i duplicati. Il comando base dell'SQL:

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

dove le parentesi quadre indicano che quella parte di codice è opzionale, mentre, le parentesi graffe indicano che quella parte può essere assente oppure può essere ripetuto tante volte. Un attributo nome di una tabella "Studenti S" si può denotare come: *nome, Studenti.nome oppure S.nome*. Il primo caso non può essere utilizzato se un'altra tabella ha lo stesso attributo. La **lista degli attributi** ha questa sintassi:

```
Attributi ::= *
 | Expr [[AS] Nuovonome] {, Expr [[AS] Nuovonome]}

Expr ::= [Ide.]Attributo | Const
|  ( Expr )  |  [-] Expr [Op Expr]
|  COUNT(*)
|  AggrFun ( [DISTINCT] [Ide.]Attributo)
```

Oltre alle espressioni elementari, vi sono anche le espressioni aggregate in cui si possono usare gli operatori come *SUM, COUNT, AVG, MAX, MIN*. La clausola DISTICT è utile con l'operatore COUNT:

SELECT

- **COUNT ***: l'asterisco indica l'intera riga, e quindi conta le righe all'interno della tabella studenti
- **COUNT email**: conta il numero di righe in cui email è diverso da NULL
- **COUNT DISTICT email**: conta il numero di righe in cui email è diverso da NULL ed elimina i duplicati

FROM Studenti

Partendo dalle tabelle Studenti ed Esami:

```
SELECT S.nome, E.data
FROM Studenti S, Esame E
WHERE E.matricola = S.matricola AND E.materia = 'BD' AND E.voto = 30
```

cioè sono le righe in cui *E.matricola* è uguale ad *S.matricola*, e in cui la materia è uguale a BD e voto uguale a 30, di queste righe si vuole conoscere nome dello studente e la data dell'esame. La lista delle tabelle nella sua forma più semplice è *FROM Studenti S, Esami E*, ma nella clausola FROM oltre a operare un prodotto cartesiano si può anche utilizzare la giunzione specificando la sintassi **JOIN - ON**, per esempio: *Studenti S JOIN Esami e ON S.matricola = E.matricola*. In alternativa *Studenti S JOIN Esami E* ***USING(matricola)***. La **condizione** è la combinazione booleana di predicati tra cui:

- Expr Comp Expr
- Expr Comp (Sottoselect che torna un valore)
- [NOT] EXISTS (Sottoselect)
- Espr Comp (ANY | ALL) (Sottoselect)
- Expr [NOT] IN (Sottoselect) (oppure IN (v1,..,vn))
- Comp: <, =, >, <>, <=, >=

Se il risultato finale lo si vuole ordinato si aggiunge la clausola **ORDER BY**. Tramite l'operatore **EXISTS** si esprime la quantificazione esistenziale, mentre tramite **NOT EXISTS** si esprime la quantificazione universale. Un altro modo per indicare la quantificazione esistenziale o universale è il **confronto quantificato**, se si scrive 3 = **ANY** {**SET**} significa che $\exists$ x $\in$ Set . 3 = x, mentre 3 = **ALL** {**SET**} rappresenta $\forall$ x $\in$ Set . 3 = x. Spesso ANY viene abbreviato con **IN**, e <> ALL con **NOT IN**. L'ultimo modo per esprimere la quantificazione universale è la **giunzione esterna**: la giunzione ha un effetto moltiplicativo nel senso che alcune righe le moltiplica e altre le cancella.

R

A	B
1	a
2	b
3	c

S

A	C
1	x
3	y
5	z

A	B	C
1	a	x
3	c	y

Il fatto di perdere qualche riga della tabella sinistra, in qualche situazione, non è gradito e per risolvere questo problema è stata definita l'operatore di **LEFT OUTER JOIN**.

R

A	B
1	a
2	b
3	c

S

A	C
1	x
3	y
5	z

A	B	C
1	a	x
2	b	
3	c	y

Nella colonna C, visto che non esiste nessun elemento della tabella S che si potesse connettere agli elementi della tabella R, viene inserito un valore NULL. Esiste anche una nozione simmetrica di **RIGHT OUTER JOIN** dove in questo caso si recuperano le righe perse della tabella di destra e si inseriscono dei NULL nelle colonne che provengono dalla tabella di sinistra.

R

A	B
1	a
2	b
3	c

S

A	C
1	x
3	y
5	z

A	B	C
1	a	x
3	c	y
5		z

Infine esiste anche il **FULL OUTER JOIN** dove si recuperano le righe perdute della tabella di sinistra e di destra.

R

A	B
1	a
2	b
3	c

S

A	C
1	x
3	y
5	z

A	B	C
1	a	x
2	b	
3	c	y
5		z

SQL fornisce il valore speciale **NULL** che può essere presente in qualunque campo di qualunque ennupla a meno che non sia specificato un vincolo **NOT NULL** o *primary key* per quel campo. NULL ha l'aspetto particolare che appartiene a tutti i tipi (intero, bool, data, ...), e fondamentalmente indica due situazioni generali: un campo in cui non si conosce un valore o non si applica. Ad esempio se nel campo telefono di studente è presente il valore NULL può significare che non si conosce il numero di cellulare o che lo studente non possiede un telefono, e quindi il valore non si applica. La presenza del NULL introduce dei problemi: se il campo telefono di una persona è NULL e si vuole sapere se il numero di un studente è 1027950, SQL risponde con **unknown**, ma se si volesse trovare tutti gli studenti in cui numero di telefono è uguale a NULL, non si può scrivere nel WHERE *telefono = NULL* perchè non restituisce nessun studente come risultato, per questo motivo è stato definito un operatore **IS NULL** che ritorna TRUE quando il valore è NULL e FALSE quando il valore non è NULL. Le funzioni di aggregazione ignorano il NULL, cioè se si chiedesse un MIN{3, NULL, 5} la risposta sarebbe 3, il MIN ritorna NULL solamente nel caso in cui tutti i valori sono uguali a NULL, in modo analogo COUNT{13, 15, NULL, 5} = 3, se si volesse contare anche il NULL, si deve usare COUNT *.

Esercizio 1. Si consideri il seguente schema relazionale: Aule(idA, nomeAula, edificio, capienza) AuleCorsi(idA*, idC*, ora, giorno) Corsi(idC, nomeCorso, annoAccademico, numStudenti, docente)

- *Per ogni lezione in cui il giorno è "Giovedi" si indichi nome aula, ora, nome-corso*

```
SELECT A.nomeAula, C.nomeCorso, AC.ora
FROM Aule A
JOIN AuleCorsi AC ON (A.idA = AC.idA)
JOIN Corsi C ON (AC.idC = C.idC)
WHERE AC.giorno = 'giovedi'
```

- *Per ogni corso che hanno almeno una lezione il giovedì, si indichi nome-corso e docente*

```
SELECT C.nomeCorso, C.docente
FROM Corsi C
JOIN AuleCorsi AC ON (AC.idC = C.idC)
WHERE AC.giorno = 'giovedi'
```

  un'altra soluzione può essere:

```
SELECT C.nomeCorso, C.docente
FROM Corsi C
WHERE EXISTS(SELECT *
             FROM AuleCorsi AC
             WHERE C.IdC = AC.Idc AND AC.giorno = 'giovedi')
```

  che significa che per ogni corso valuta l'insieme di tutte le aule-corsi legate al corso che hanno lezione giovedì e se ne esiste qualcuna si interrompe la valutazione del sottoselect, l'EXISTS ritorna true, e si riporta quel corso nel risultato. In realtà queste due query possono avere un risultato leggermente diverso nel caso in cui un corso abbia due lezioni il giovedì: la prima query restituisce due volte il risultato docente e nomeCorso, mentre la seconda query una volta sola. Per fare in modo che la prima query restituisca lo stesso risultato della seconda query si deve aggiungere DISTINCT. Questi sono due modi diversi per fare la quantificazione esistenziale: tramite giunzione o tramite clausola EXISTS ;

- *Per ogni corso che ha solo lezione il giovedì, si indichi nome-corso e docente*

```
SELECT C.nomeCorso, C.docente
FROM Corsi C
WHERE NOT EXISTS(SELECT *
                 FROM AuleCorsi AC
                 WHERE c.IdC = AC.Idc AND NOT(AC.giorno = 'giovedi'))
```

  non avendo un quantificatore per $\forall$, lo si deve riscrivere come $\neg \exists \neg$ per dualità di De Morgan;

- *Per ogni aula in cui c'è almeno un corso con più di 100 studenti, nome aula*

```
SELECT DISTICT A.nomeAula
FROM Aule A
JOIN AuleCorsi AC ON(A.idA = AC.idA)
JOIN Corsi C ON (AC.idC = C.idC)
WHERE C.numStud >= 100
```

  un altro modo è:

```sql
SELECT A.nomeAula
FROM Aule A
WHERE EXISTS (SELECT *
              JOIN AuleCorsi AC
              JOIN Corsi C ON (AC.idC = C.idC)
              WHERE A.idA = AC.IdA AND C.numStud >= 100)
```

Il raggruppamento è una operazione di analisi che permette di verificare come determinate misure dipendono da determinate dimensioni, per esempio *Per ogni materia, trovare nome della materia e voto medio*

```sql
SELECT E.materia, AVG(E.voto)
FROM Esami E
GROUP BY E.materia
```

dato che il voto medio dipende da una dimensione che è la materia, si usa la **GROUP BY** per raggruppare rispetto alla materia e per ogni materia calcola il voto medio. *Per ogni studente, trovare nome e voto medio:*

```sql
SELECT S.nome, AVG(E.voto)
FROM Studenti S, Esami E
WHERE S.matricola = E.matricola
GROUP BY S.matricola, S.nome
```

dato che si vuole raggruppare per ogni studente, S.matricola deve essere inserito all'interno della GROUP BY, ma dato che interessa vedere anche il nome si deve aggiungere obbligatoriamente anche S.nome. In presenza del GROUP BY si può aggiungere la clausola **HAVING** che può contenere MIN, COUNT, ... ma per tutte le dimensioni che sono inserite in HAVING devono essere già presenti nella GROUP BY.

Esercizio 2. Si consideri il seguente schema relazionale: Aule(idA, nomeAula, edificio, capienza), AuleCorsi(idA*, idC*, ora, giorno), Corsi(idC, nomeCorso, annoAccademico, numStudenti, idD*), Docenti(idD, nome, cognome, dipartimento)

- *Per ogni dipartimento con più di 30 docenti, riportare il nome del dipartimento e il numero di docenti che vi appartengono*

```sql
SELECT D.dipartimento, COUNT(*)
FROM Docenti D
GROUP BY D.dipartimento
HAVING COUNT(*) > 30
```

- *Per ogni dipartimento, riportare il nome del dipartimento, il numero di docenti che vi appartengono, il numero di corsi che vengono insegnati dai docenti di quel dipartimento, e il numero totali di studenti di tali corsi*

```sql
SELECT D.dipartimento, COUNT(*), COUNT(DISTICT D.idD), SUM(C.numStudenti)
FROM Docenti D JOIN Corsi C ON (C.idD = D.idD)
GROUP BY D.dipartimento
HAVING COUNT(*) > 30
```

dove *COUNT(DISTICT d.idD)* permette di contare un singolo docente una volta sola anche se quel docente tiene molti corsi. Un altro modo è creare una vista:

```sql
CREATE VIEW docentiPerDipartimento                    -- la vista associa ad ogni dipartimento
                                                         il numero di docenti
AS (SELECT D.Dipartimento, COUNT(*) AS numDocenti
    FROM Docenti D
    GROUP BY D.dipartimento)

SELECT D.dipartimento, COUNT(*), SUM(C.numStudenti), DPD.numDocenti
FROM Docenti D JOIN Corsi C ON (C.idD = D.idD)        -- questa giunzione serve
                                                         per contare il numero dei corsi
                                                         e sommare il numero degli studenti
JOIN docentiPerDipartimento DPD ON (D.Dipartimento = DPD.Dipartimento)
GROUP BY D.Dipartimento, DPD.numDocenti
```

con **CREATE VIEW** si va a creare una tabella virtuale, ed è una interrogazione a cui gli si da un nome (*docentiPerDipartimento*. Dopo aver creato una vista la si può usare all'interno delle clausole FROM;

- *Per ogni docente che tiene solo corsi con più di 100 studenti riportare il nome del docente e il suo idD*

```sql
SELECT D.idD, D.nome
FROM Docenti D
WHERE NOT EXISTS (SELECT *
                  FROM Corsi C
                  WHERE C.idD = D.idD AND NOT (C.numStudenti >= 100)
```

cioè non esiste un corso insegnato dal docente D in cui il numero di studenti non è maggiore di 100.

- *Per ogni coppia di docenti con lo stesso cognome, riportare nome, cognome e il dipartimento del primo, e il nome, cognome e il dipartimento del secondo*

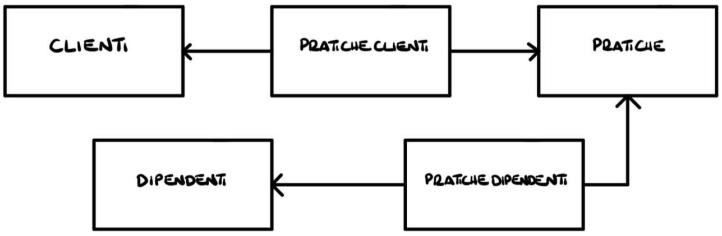
```
SELECT D1.nome, D1.cognome, D1.dipartimento,
       D2.nome, D2.cognome, D2.dipartimento
FROM Docenti D1, Docenti D2
WHERE D1.cognome = D2.cognome AND D1.idD <> D2.idD
```

- *Per i corsi che fanno lezione solo in aule con capienza maggiore di 100, visualizzare idC e nome del corso*

```
SELECT C.idC, C.nomeCorso
FROM Corsi C
WHERE NOT EXIST (SELECT *
                 FROM Aule A, AuleCorsi AC
                 WHERE A.idA = AC.idA AND NOT (A.capienza > 100))
```

se si sceglie di inserire AuleCorsi all'interno del primo FROM la query è esistenziale, mentre se si inserisce all'interno del secondo FROM la query è universale

Esercizio 3. Dato il seguente schema relazionale che descrive pratiche associate ciascuna a più clienti e seguita ciascuna da più dipendenti: Clienti(idC, nome, cognome, nazione), PraticheClienti(idP*, idC*), Pratiche(idP, data, nomeBreve, descrizione, budget), PraticheDipendenti(idP*, idD*), Dipendenti(idD, nomeD, cognomeD, annoAssunzione), rispondere alle seguenti interrogazioni:



- *Per tutte le pratiche che sono associate anche a clienti italiani, riportare idP e nome breve*

```
SELECT DISTICT P.idP, P.nomeBreve
FROM Pratiche P
JOIN PraticheClienti PC ON (P.idP = PC.idP)
JOIN Clienti C ON (C.idC = PC.idC)
WHERE C.nazione = 'Italia'
```

è fondamentale aggiungere DISTICT, altrimenti una pratica connessa a tanti clienti italiani viene ripetuta tante volte. In alternativa si può scrivere:

```
SELECT P.idP, P.nomeBreve
FROM Pratiche P
WHERE EXIST (SELECT *
             FROM PraticheClienti PC
             JOIN Clienti C ON (C.idC = PC.idC)
             WHERE P.idP = PC.idP AND C.nazione = 'Italia')
```

se si vuole evitare di scrivere il JOIN:

```
SELECT P.idP, P.nomeBreve
FROM Pratiche P
WHERE EXISTS (SELECT *
             FROM PraticheClienti PC, Clienti C
             WHERE P.idP = PC.idP AND C.idC = PC.idC AND C.nazione = 'Italia')
```

- *Per tutte le pratiche che sono associate solo a clienti italiani, riportare idP e nome breve*

```
SELECT P.idP, P.nomeBreve
FROM Pratiche P
WHERE NOT EXISTS (SELECT *
                  FROM PraticheClienti PC
                  JOIN Clienti C ON (C.idC = PC.idC)
                  WHERE P.idP = PC.idP AND NOT (C.nazione = 'Italia'))
```

- *Per tutte le pratiche in cui tutti i clienti associati appartengono alla nazione N, riportare idP, nome breve*

```sql
SELECT P.idP, P.nomeBreve
FROM Pratiche P
WHERE NOT EXISTS (SELECT *
                  FROM PraticheClienti PC1, PraticheClienti PC2, Clienti C1, Clienti C2
                  WHERE P.idP = CP1.idP AND P.idP = CP2.idP
                  AND C1.idC = CP1.idC AND C2.idC = CP2.idC
                  AND C1.nazione <> C2.nazione)
```

cioè per dire che tutti i clienti appartengono alla stessa nazione si può scrivere che non esistono due clienti originari da due nazioni diverse. Si è aggiunto PC1 e PC2 perchè per transitività scrivere solamente *C1.idC = PC.idC AND c2.idC = PC.idC* significa che C1 e C2 sono lo stesso cliente, quindi non sarebbe possibile avere due nazioni diverse, e quindi la query interna restituirebbe sempre l'insieme vuoto. Scriverlo con il JOIN verrebbe:

```sql
SELECT P.idP, P.nomeBreve
FROM Pratiche P
WHERE NOT EXISTS (SELECT *
                  FROM PraticheClienti PC1, PraticheClienti PC2
                  JOIN Clienti C1 ON (PC1.idC = C1.idC)
                  JOIN Clienti C2 ON (PC2.idC = C2.idC)
                  WHERE P.idP = CP1.idP AND P.idP = PC2.idP
                  AND C1.nazione <> C2.nazione)
```

- *Per tutte le pratiche in cui tutti i clienti associati appartengono alla nazione N, riportare idP, nome breve e la nazione N*

```sql
SELECT P.idP, P.nomeBreve, C1.nazione
FROM Pratiche P
JOIN PraticheClienti PC1 ON (P.idP = PC.idP)
JOIN Clienti C1 ON (PC.idC = C1.idC)
WHERE NOT EXISTS (SELECT *
                  FROM PraticheClienti PC2
                  JOIN Clienti C2 ON (PC2.idC = C2.idC)
                  WHERE P.idP = CP2.idP AND C1.nazione <> C2.nazione)
```

cioè esternamente si effettua una giunzione che serve per verificare che una pratica abbia almeno un cliente e scopre quale è la sua nazione, mentre, internamente si verifica che non esistano due clienti di due nazioni diverse

- *Per ogni nazione, il budget massimo tra i progetti che hanno almeno un cliente*

```sql
SELECT C.nazione, MAX(P.budget)
FROM Clienti C
JOIN PraticheClienti PC ON (C.idC = PC.idC)
JOIN Pratiche P ON (P.idP = PC.idP)
GROUP BY C.nazione
```

- *Per ogni cliente, idC, nome, cognome e numero di pratiche in cui è coivolto quel cliente che abbiano il budget > 1000*

```sql
SELECT C.idC, C.nome, C.cognome, COUNT(P.idP)
FROM Clienti C
JOIN PraticheClienti PC ON (C.idC = PC.idC)
JOIN Pratiche P ON (P.idP = PC.idP)
WHERE P.budget > 1000
GROUP BY C.idC, C.nome, C.cognome
```

- *Per ogni dipendente che segue solo pratiche in cui almeno un cliente ha nazione = 'italia', restituire idD, nome e cognome*

```sql
SELECT D.idD, D.nome, D.cognome
FROM Dipendenti D
WHERE NOT EXISTS (SELECT *
                  FROM PraticheDipendenti PD JOIN Pratiche P ON (PD.idP = P.idP)
                  WHERE PD.idD = D.idD AND NOT EXISTS (SELECT *
                                                       FROM PraticheClienti PC
                                                       JOIN Clienti C ON (PC.idC = C.idC)
                                                       WHERE PC.idP = P.idP
                                                       AND C.nazione = 'Italia'))
```

dal punto di vista logico: *D.idD, D.nome, D.cognome per tutti quei dipendenti D tali che per ogni pratica P seguita da D esiste un cliente C della pratica P tale che C.Nazione = 'italia'*

Esercizio 4. Si consideri lo schema: CorsiDiStudio(idCdS, nomeCdS, dipartimentoCdS), Studenti(idStud, idCdS*, cognomeStudente), Esami(idStud*, idMat*, voto, anno, semestre, idDoc*), Iscrizioni(idStud*, idMat*, anno), Docenti(idDoc, cognomeDocente), Materie(idMat, dipartimentoMat, nomeMateria, crediti)

- *Si trovi il nome di quegli studenti che hanno solo iscrizioni tra il 2012 e il 2015*

```
SELECT S.cognomeStudente
FROM Studenti S
JOIN Iscrizioni I ON (S.idStud = I.idStud)
WHERE NOT EXIST (SELECT *
                 FROM Iscrizioni I
                 WHERE S.IdStud = I.IdStud AND NOT (2012 <= I.anno AND I.anno <= 2015);
```

  o in alternativa:

```
SELECT S.cognomeStudente
FROM Studenti S
JOIN Iscrizioni I ON (S.idStud = I.idStud)
WHERE NOT IN (SELECT I.idStud
              FROM Iscrizioni I
              WHERE NOT (2012 <= I.anno AND I.anno <= 2015);
```

- *Per ogni dipartimento, si riporti il dipartimento e il numero totale di studenti il cui corso di studio appartiene a quel dipartimento*

```
SELECT CDS.dipartimento, COUNT(*)
FROM CorsoDiStudio JOIN Studenti USING (idCdS)
GROUP BY CDS.dipartimento
```

- *Per tutti gli studenti che hanno superato solo esami per cui la materia risulta un'iscrizione relativa allo studente, riportare idStud e cognome*

```
SELECT S.cognomeStudente, S.idStud
FROM Studenti S
WHERE NOT EXISTS (SELECT *
                  FROM Esami E JOIN Materie M USING (idMat)
                  WHERE E.idStud = S.idStud AND NOT EXISTS (SELECT *
                                                            FROM Iscrizioni I
                                                            WHERE I.idStud = S.idStud AND
                                                                  I.idMat = M.idMat))
```

  cioè gli studenti tale che per ogni esame che lo studente ha superato, se si va a guardare la materia di quell'esame allora esiste una iscrizione. In termini insiemistici:

$$\{\ \text{S.cognome, S.idStud} \mid \text{S} \in \text{Studenti} \quad \forall\ \text{Esami E di S, Materie M di E} \quad \exists\ \text{Iscrizione I che collega S ad M}\ \}$$

- *Per ogni materia e per ogni anno, riportare il voto massimo che è stato assegnato in quella materia in quell'anno nonché idStud e cognome dello studente che lo ha preso*

```
CREATE VIEW MaxVoto
AS (SELECT E.idMat, E.anno, MAX(E.voto) as MaxVoto
    FROM Esami E
    GROUP BY E.idMat, E.anno )

SELECT s.idStud, s.cognomeStudente, M.idMat, M.anno
FROM MaxVoto M JOIN Esami E USING (idMat, anno)
JOIN Studenti S ON (E.idStud = S.idStud)
WHERE M.MaxVoto = E.voto
```

  o in alternativa si può utilizzare una interrogazione universale:

```
SELECT s.idStud, s.cognomeStudente, E.idMat, E.anno
FROM Esami E
JOIN Studenti S ON (E.idStud = S.idStud)
WHERE NOT EXISTS (SELECT *
                  FROM Esami E2
                  WHERE E2.idMat = E.idMat AND E2.anno = E.anno AND E2.voto > e.voto)
```

  cioè quegli studenti per cui non esiste un altro studente che ha preso un voto strettamente maggiore. Un'altra soluzione ancora è

```sql
SELECT s.idStud, s.cognomeStudente, E.idMat, E.anno
FROM Esami E
JOIN Studenti S ON (E.idStud = S.idStud)
WHERE E.voto >= ALL (SELECT E2.voto
                     FROM Esami E2
                     WHERE E2.idMat = E.idMat AND E2.anno = E.anno)
```

ovvero gli esami per cui il voto di quell'esame E è superiore o uguale a tutti i voti di quello stesso anno e di quella stessa materia

- *Per ogni materia in cui tutti gli studenti iscritti sono dello stesso CdS, riportare il nome della materia e il nome del CdS*

```sql
SELECT M.nomeMateria, C.nomeCdS
FROM Materie M
JOIN Iscrizioni I ON (I.idMat = M.idMat)
JOIN Studenti S ON (I.idStud = S.idStud)
JOIN CorsiDiStudio C ON (S.idCdS = C.idCdS)
WHERE NOt EXISTS (SELECT *
                  FROM Iscrizioni I2
                  JOIN Studenti S2 ON (I2.idStud = S2.idStud)
                  WHERE I2.idMat = M.idMat AND S.idCdS != S2.idCdS )
```

o in alternativa

```sql
SELECT M.nomeMateria, MIN(C.nomeCds)
FROM Materie M
JOIN Iscrizioni I ON (I.idMat = M.idMat)
JOIN Studenti S ON (I.idStud = S.idStud)
JOIN CorsiDiStudio C ON (S.idCdS = C.idCdS)
GROUP BY M.nomeMateria
HAVING COUNT(DISTINCT C.nomeCdS) = 1
```

4.2 Per modificare i dati

SQL fornisce una sintassi per modificare i dati:

- **INSERT**, per inserire nuove righe. Per inserire due nuovi studenti:

```sql
INSERT INTO Studenti (nome, cognome)
VALUES ('Mario', 'Rossi'), ('Lucia', 'Bianchi')
```

La parte *(nome, cognome)* è opzionale e se non si scrive è come se si elencasse tutti gli attributi della tabella nell'ordine dichiarati. Un altro modo è:

```sql
INSERT INTO Studenti (nome, cognome)
SELECT nome, cognome
FROM Persone
WHERE annoNascita = 2001
```

cioè si vogliono iscrivere alla scuola tutte le persone nate nel 2001, spesso questa tecnica è usata quando si migrano i dati da una base di dati ad un'altra;

- **UPDATE**, per modificare le righe già presenti. Per aggiornare il voto di uno studente con matricola 101:

```sql
UPDATE Studenti
SET voto = 28
WHERE matricola = '101'
```

se si volesse raddoppiare il voto di tutti gli studenti di nome "Mario":

```sql
UPDATE Studenti
SET voto = voto * 2
WHERE nome = 'Mario'
```

- **DELETE**, per rimuovere righe. Per cancellare lo studente con matricola 101 basta scrivere:

```sql
DELETE FROM Studenti
WHERE matricola = '101'
```

Chapter 5

SQL: Per definire e amministrare basi di dati

5.1 Definizione di basi di dati

SQL non è solo un linguaggio di interrogazione ma ha anche funzionalità di definizione di basi di dati (DDL). Per definire una tabella si usa il comando **CREATE TABLE** in cui all'interno si elencano i campi e gli eventuali vincoli relativi alla tabella. Ogni campo può avere associato dei vincoli:

- **NOT NULL**, evita il campo ammette il valore NULL;
- **Check**, tutte le volte che si va a inserire una nuova riga, e quindi un nuovo valore per un certo campo, si verifica prima una certa condizione. Può essere messo accanto all'attributo a cui si riferisce oppure in fondo;
- **Default**, quando non si inserisce un valore per un campo che possiede il vincolo default prenderà un valore prestabilito.
- **PRIMARY KEY**, viene assegnato anche un nome, nell'esempio *pk-impiegato*, che serve soprattutto per i messaggi di errore;
- **FOREIGN KEY**, è un vincolo che ha bisogno fondamentalmente di 2 informazioni: qual è il campo e verso quale tabella fa riferimento. Nel esempio, il campo supervisore è chiave esterna verso la chiave primaria della tabella impiegati, cioè ha un campo supervisore che è chiave esterna verso la tabella impiegati stessa. La chiave esterna ha senso solo se fa riferimento a una tabella che possiede una chiave primaria.

```
CREATE TABLE  Nome
        ( Attributo  Tipo [ValoreDefault] [VincoloAttributo]
          {, Attributo  Tipo [Default] [VincoloAttributo]}
          {, VincoloTabella})
```

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

Ciò che si crea con un CREATE si può eliminare con il comando **DROP**. Si immagina di avere una tabella Esami con campo candidato che è chiave esterna verso il campo matricola della tabella Studenti. Ci sono due modi per violare il vincolo di chiave esterna:

- inserendo un candidato con matricola 110 che non esiste nella tabella degli studenti;
- tramite cancellazione, andando a eliminare lo studente con matricola 110 dopo che esistono 3 verifiche nella tabella Esami sostenuti dallo studente.

Esistono tre possibilità quando un utente prova a cancellare uno studente che ha superato esami:

- **NO ACTION**, possibilità di default che proibisce l'azione;
- **CASCADE**, esistono dei casi in cui il vincolo di FOREIGN KEY è debole. Per esempio se l'esame interessa solamente se lo studente risiede nella base di dati, nel momento in cui si cancella lo studente si può fare una cancellazione in cascata di tutti i suoi esami;
- **SET NULL**, quando si cancella lo studente, si tengono gli esami ma mettendo a NULL il campo studente.

Dopo aver creato una tabella e dopo aver iniziato ad aggiungere valori, si possono inserire colonne con il comando **ALTER TABLE Nome ADD COLUMN NuovoAttr Tipo** dove il nuovo attributo assume il valore NULL in tutte le colonne già esistenti, che si potranno volendo modificare con il comando UPDATE. Un altro modo per creare una tabella è copiarla da una espressione SELECT.

```
CREATE TABLE Supervisori
        SELECT  Codice, Nome, Qualifica, Stipendio
        FROM  Impiegati
        WHERE Supervisore IS NULL
```

5.2 Viste

Un altro concetto molto importante sono le **viste**:

```
CREATE VIEW Nome [(Attributo {, Attributo})]
    AS EspressioneSELECT [WITH CHECK OPTION];
```

```
CREATE VIEW Supervisori
    AS      SELECT  Codice, Nome, Qual., Stip.
            FROM    Impiegati
            WHERE   Supervisore IS NULL
```

La CREATE TABLE è statica cioè se si crea la tabella Supervisori come nell'esempio e subito dopo si modifica la tabella Impiegati inserendo un nuovo impiegato con supervisore IS NULL, la tabella Supervisori non cambia, mentre CREATE VIEW è dinamica, ovvero ogni volta che si aggiunge o si rimuove un impiegato che soddisfa la query, la vista cambia. Le viste hanno una grande utilità e sono utilizzate per scopi diversi:

- indipendenza logica, cioè nascondere certe modifiche all'organizzazione logica dei dati;
- per offrire visioni diverse degli stessi dati senza ricorrere a duplicazioni;
- per rendere più semplici, o per rendere possibili, alcune interrogazioni. Per esempio: *trovare il numero medio di impiegati dei dipartimenti*

```
1 CREATE VIEW NumImpiegatiDip(Dipart.,NumImp)
2 AS (SELECT Dipartimento, COUNT(*)
3     FROM Impiegati
4     GROUP BY Dipartimenti)
5
6 SELECT AVG(NumImp)
7 FROM NumImpiegatiDip
```

che altrimenti non si può scrivere senza view perchè AVG(COUNT(*) non è un comando ammesso.

Nello schema della base di dati oltre a gestire tabelle e metadati si possono memorizzare **procedure/funzioni**, questo permette di:

- definire dei **tricker** cioè un pezzo di codice associato ad un evento e a una condizione, tale che tutte le volte che si verifica l'evento ed è vera la condizione, il DBMS esegue automaticamente il codice senza l'intervento dell'utente. I trigger possono servire per trattare i vincoli non esprimibili nello schema, ma possiedono un grosso svantaggio che possono mandare il DBMS in uno stato di loop infinito: se il trigger A esegue una azione che invoca un trigger B, e il trigger B fa una azione che causa l'invocazione del trigger A. Il loop infinito non manda in crash il sistema, ma causa un rallentamento;
- utilizzare dei meccanismi di protezione della base di dati per determinare i diritti di esecuzione di queste procedure/funzioni, cioè le associa a certi utenti e solamente loro hanno diritto a eseguire queste procedure o funzioni.

```
CREATE FUNCTION contaStudenti IS
  DECLARE
   tot INTEGER;
  BEGIN
   SELECT COUNT(*) INTO tot FROM STUDENTI;
   RETURN (tot);
  END
```

```
CREATE TRIGGER Nome
      PrimaODopoDi  Evento {, Evento}
      ON Tabella [WHEN Condizione]
      [Granularità]
      Azione
  PrimaODopoDi  := BEFORE | AFTER
  Evento := INSERT | DELETE | UPDATE OF Attributi
  Granularità :=  FOR EACH ROW | FOR EACH STATEMENT
```

5.3 Controllo degli accessi

Chi crea lo schema della base di dati è l'unico che può eseguire i comandi CREATE, ALTER e DROP, ma normalmente si desidera assegnare i diritti di accesso a determinati utenti. I privilegi si possono concedere ad utenti che corrispondono a persone fisiche, ad utenti che coincidono, dal punto di vista logico, a dei gruppi di utenti, e da entità che corrispondo a del codice. L'utente che riceve il privilegio può avere il diritto di cedere a sua volta i diritti ad altri e quindi possiede l'opzione **GRANT**, oppure no. I privilegi possono essere:

- **SELECT**, lettura dei dati;
- **INSERT**, inserire record;
- **DELETE**, cancellazione di record;
- **UPDATE**, modificare record o solo attributi;
- **REFERENCES**, definire chiave esterna che fa riferimento ad un attributo di una tabella che magari non si hanno i diritti.

I privilegi si possono revocare con il comando **REVOKE** e si può fare in cascata eliminando tutti i diritti, si può revocare solo la GRANT OPTION, e si può revocare solo determinati diritti. Un **indice** della tabella impiegati, sull'attributo nome è una struttura dati che si va a memorizzare oltre alla tabella impiegati, ed è una tabella composta da due campi: il nome e la lista di record. Vicino al primo campo, per esempio "Andrea", ci sarà la lista di tutti i record che hanno il nome "Andrea", e in più ci sarà un albero di ricerca che permette rapidamente, dato un nome, di trovare tutti i puntatori a tutti i record nella tabella in cui il nome è "Andrea". L'indice è una struttura dati costosa da mantenere, e rallenta l'inserimento, ma permette di velocizzare le ricerche senza scorrere tutta la tabella. Tutte le tabelle che si creano usando il DDL viene memorizzato in strutture dati interne al DBMS che sono accessibili esternamente tramite delle viste scrivendo query in SQL. Questo insieme di dati viene chiamato **catalogo**, e possiede i propri diritti di accesso.

Chapter 6

SQL: per programmare le applicazioni

6.1 Approcci

Ci sono 3 diversi approcci allo scopo di scrivere del codice che contiene al suo interno l'SQL:

- **Linguaggio integrato**, si basa su un linguaggio unico. Tipicamente un produttore di DBMS inventa un linguaggio costruito ad-hoc per lo scopo, quindi è un linguaggio che contiene l'SQL come sottolinguaggio ma aggiunge tutte quelle caratteristiche che mancano all'SQL per essere un linguaggio completo (if - else, for, ...);
- **Linguaggio convenzionale + API**, usa un linguaggio convenzionale noto (C, Python, ...) e una libreria di funzioni del tipo "accedi alla base di dati", "spedisci questa stringa di SQL alla base di dati", ... La differenza fondamentale tra i due linguaggi è che che quando si programma con il linguaggio integrato si sta dialogando con un compilatore che conosce il linguaggio SQL e lo schema del DBMS, quindi se nelle query SQL ci fosse qualche problema il compilatore avvisa il programmatore immediatamente. Invece quando si programma con un linguaggio convenzionale, la stringa SQL per il compilatore è semplicemente una stringa e quindi non la controlla;
- **Linguaggio che ospita l'SQL**, è un approccio intermedio, vi è linguaggio convenzionale, un meccanismo che permette di inserire delle parti di SQL all'interno del linguaggio e un precompilatore che controlla i comandi di SQL, li sostituisce con chiamate a funzioni predefinite, e genera un programma nel linguaggio convenzionale + API.

6.2 Linguaggio integrato

Il linguaggio **PL/SQL** di Oracle, uno dei più celebri, contiene tutti gli operatori tipici dei linguaggi di programmazione e dell'SQL, e permette:

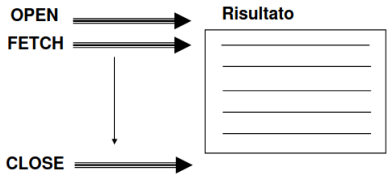
- di definire variabili che possono essere di tipo scalare, record (annidato), insieme di scalari, insieme di record e cursore;
- di definire i tipi delle variabili andando a copiare i tipi della base di dati;
- di eseguire interrogazioni SQL ed esplorarne il risultato;
- di modificare la base di dati;
- di definire procedure e moduli;
- di gestire il flusso del controllo, le transazioni, le eccezioni;

```
CREATE
PROCEDURE Esempio1(
     p_Mat IN Studenti.Matricola%TYPE) IS
DECLARE
--   Identificatori per lo scambio dati
   XNome CHAR;
   XProvincia Studenti.Provincia%TYPE;
   XAttributi Studenti%ROWTYPE;
   prv_manca EXCEPTION;
BEGIN
-- ricerca di ennupla : stampa Nome e Provincia
SELECT  Nome, Provincia INTO XNome, XProvincia
FROM  Studenti WHERE  Matricola = p_Mat;
IF XProvincia IS NULL THEN
    RAISE prv_manca
    ELSE PRINT ….
END IF
EXCEPTION
    WHEN  prv_manca THEN  <gestione eccezione>
END;
```

6.2.1 Cursore

Un cursore permette di analizzare tanti record, uno alla volta. Un cursore viene definito con un'espressione SQL, poi si manda un comando open per far calcolare al DBMS il risultato e in seguito con un opportuno comando si trasferiscono i campi delle ennuple in opportune variabili del programma. In PL/SQL le operazioni che si possono fare con un cursore sono:

- **Open**, per eseguire la query, memorizzare il risultato in un'area di memoria;
- **Fetch**, per leggere uno dopo l'altro tutti i record del risultato;
- **Close**, per finire di analizzare i dati e quindi si rilascia l'area di memoria.



6.3 Linguaggi con interfaccia API

Invece di modificare il compilatore di un linguaggio, si usa una libreria di funzioni/oggetti che permette di comunicare con il DBMS. Queste librerie sono "abbastanza" indipendenti dal DBMS. In questo modo tutte le stringhe sono la stessa cosa, per cui il compilatore non controlla le stringhe SQL, e un altro svantaggio è che "scrivere del codice che genera codice" è complicato per via delle gestione delle stringhe (gli apici). Il vantaggio di usare questo approccio è che lo sviluppatore può programmare in Java anzichè in PL/SQL, e usare l'ambiente di programmazione Java preferito. L'ultimo vantaggio è la flessibilità cioè, in PL/SQL si possono parametrizzare solamente le costanti, invece con un linguaggio con interfaccia API si può parametrizzare qualsiasi cosa.

```
class StampaNomiStudenti{
public static void main(String argv[]){
Class.forName("driver per DBMS");
Connection con = // connect
   DriverManager.getConnection("url", "login", "pass");
Statement stmt = con.createStatement(); // crea un oggetto per comando SQL
String query = "SELECT Nome
                FROM Studenti WHERE Provincia =\'" + argv[0] + " \'";
ResultSet iter = stmt.executeQuery(query);
System.out.println("Nomi trovati:");
try { // gestore eccezioni
      // ciclo sul risultato
    while (iter.next()) {
        String nome = iter.getString("Nome");
        int anno = iter.getInt("AnnoNascita");
        System.out.println(" Nome: " + nome + "; AnnoNascita: " + anno);
    }
} catch(SQLException ex) {
    System.out.println(ex.getMessage()+ex.getSQLState()+ex.getErrorCode());
}
stmt.close(); con.close();
}}
```

6.4 Linguaggio che ospita l'SQL

Si parte da un linguaggio, per esempio C, e all'interno di questo linguaggio si permette di inserire dei comandi SQL, oppure di dichiarare delle varabili che serviranno ad aprire un canale di comunicazione tra l'SQL e il linguaggio, cioè dichiarando delle variabili in determinate sezioni, il compilatore li renderà accessibili sia al codice C che al codice SQL. Con *EXEC SQL DECLARE* definire un cursore. Con questo approccio non si deve imparare un nuovo linguaggio di programmazione, ma basta imparare 3 comandi EXEC SQL

```
char SQLSTATE[6];
EXEC SQL BEGIN DECLARE SECTION
char c_sname[20]; short c_annoNascita;
EXEC SQL END DECLARE SECTION
short c_Provincia = "Pisa";
EXEC SQL DECLARE sinfo CURSOR FOR
  SELECT S.nome, S.annoNascita
  FROM Studenti S
  WHERE S.Provincia = :c_Provincia
  ORDER BY S.nome;
do {
  EXEC SQL FETCH sinfo INTO :c_snome, :c_annoNascita;
  printf("Nome:%s; AnnoNascita: %s \n ", c_snome,
  c_annoNascita);
} while (SQLSTATE != '02000');
EXEC SQL CLOSE sinfo;
```

6.5 Svantaggi comuni

Molto spesso l'integrazione tra la parte SQL, e la parte C, Java, ... ha lo svantaggio che l'SQL manipola insiemi, mentre il linguaggio C manipola un record per volta e quindi si deve passare tra meccanismi complessi come il cursore per fare da ponte tra queste due realtà. Un'altra difficoltà è a livello di tipi elementari, spesso il tipo intero di SQL non coincide con il tipo intero di Java, perchè il primo include anche il valore NULL a differenza del tipo intero del secondo linguaggio. A questo punto tutte le volte che si vuole leggere un intero dalla base di dati, si dovrebbe sempre prima testare se quell'intero è NULL o no.

Chapter 7

Normalizzazione

7.1 Problema

Quando si costruiscono base di dati, queste nascono da evoluzioni di una tabella contenenti molte informazioni provenienti da entità non disgiunte. Questa tabella ha una serie di problemi, la teoria che si andrà a sviluppare, cercherà di individuare la sorgente di tutti questi problemi e darà degli strumenti per correggerli.

N Inv	Stanza	Resp	Oggetto	Produttore	Descrizione
1012	256	Ghelli	Mac Mini	Apple	Personal Comp
1015	312	Albano	Dell XPS M1330	Dell	Notebook 2 GHZ
1034	256	Ghelli	Dell XPS M1330	Dell	Notebook 2GB
1112	288	Leoni	Mac Mini 2	Apple	Personal Comp

7.2 Teoria relazionale

Esistono due modi per produrre uno schema relazionale:

- Produrre un buon schema a oggetti, per poi tradurlo in uno schema relazionale;
- Partire da uno schema relazionale già esistente, dove ci possono essere già molti problemi, e cercare di modificarlo gradualmente per risolvere tutte le difficoltà.

La teoria delle progettazione relazionale studia cosa sono le ”anomalie” e come eliminarle. La teoria relazionale ha anche una grande importanza per chi deve disegnare un ottimizzatore.

Esempio. *StudentiEdEsami(matricola, nome, provincia, annoNascita, materia, voto)*
Si immagina che tutti gli esami si memorizzano in una tabella StudentiEdEsami in cui si memorizzano i dati dello studente e poi quelli dell'esame. Questo modo di lavorare non è corretto fondamentalmente perchè presenta **ridondanze**: la prima volta che viene scritta la matricola, gli altri dati dello studente sono utili, ma dalla seconda volta in poi in cui si scrive la stessa matricola, gli altri dati diventano ridondanti. La ridondanza produce **potenziali inconsistenze**: a furia di scrivere ogni volta l'annoNascita dello studente può capitare che una volta lo si inserisca male, o se si vuole aggiornare una informazione relativa a quello studente, è molto probabile che lo si faccia in un punto solo, dimenticandosi di tutte le altre copie ridondanti. Un altro problema è che se si vuole utilizzare la tabella StudentiEdEsami per gestire contemporaneamente Studenti ed Esami, allora nel momento dell'inserzione di uno studente, i campi materia e voto saranno NULL, e la prima volta che quello studente farà un esame si sostituiranno questi campi. Quindi c'è una anomalia nella prima inserzione perchè per il primo esame si userà un comando UPDATE, dal secondo esame in poi si userà INSERT. È presente lo stesso problema in fase di eliminazione perchè quando si inserisce un esame per errore e lo si vuole cancellare, invece di mettere a NULL i campi materia e voto, può capitare di cancellare l'intera riga, perdendo tutte le informazioni sullo studente. Se si vuole risolvere il problema, basta spezzare lo schema in due: una tabella per gli studenti, e uno per gli esami, dove in quest'ultima è presente una chiave esterna che permette di ricostruire la connessione con la tabella degli studenti.

7.3 Dipendenze funzionali

Una dipendenza funzionale è una descrizione formale di un fatto semantico e sono un modo per descrivere le istanze valide, cioè rientrano tra le categorie dei vincoli: $X \rightarrow Y$, il vincolo X determina Y, vuole dire che per $\forall r$ istanza valida di una relazione R, se si trovano due righe t1 e t2 dove il campo X ha lo stesso valore allora anche il campo Y deve avere lo stesso valore

MATRICOLA	NOME	...
100	MARIO	...
100	LUCIA	...
...	...	...

MATRICOLA	NOME	...
100	MARIO	...
101	LUCIA	...
100	MARIO	...

cioè se nel campo X, esistono due righe t1 e t2 con lo stesso valore (100) allora anche il campo Y di queste due righe devono avere lo stesso valore (Mario).

Esempio. *DotazioniLibri(codiceLibro, nomeNegozio, indNegozio, titolo, quantità)*
Questa tabella è ricca di ridondanza perchè se vi sono 20 negozi, e un certo libro è presente in tutti i negozi, si ripete il codice, il titolo, la quantità per tutti e 20 i negozi, e se un libro è presente 50 volte in un negozio, si ripeterà indNegozio, nomeNegozio dello stesso negozio 50 volte. Dipendenze funzionali: il codice del libro determina il titolo, il nome del negozio determina l'indirizzo e infine il codice del libro e il nome del negozio determinano l'indirizzo del negozio, il titolo e la quantità.

7.3.1 Esprimere le dipendenze funzionali

Si considera `nomeNegozio` $\Rightarrow$ `indNegozio`, ci sono tanti modi, tutti equivalenti fra di loro, di descrivere una dipendenza funzionale:

- **Espressione diretta**, se in due righe il nome del negozio è uguale allora anche l'indirizzo è uguale;
- **Per contrapposizione**, quando l'indirizzo negozio è diverso allora il nome del negozio deve essere diverso;
- **Per assurdo**, non possono esserci due righe in cui è presente lo stesso nome del negozio ma un indirizzo diverso.

Esercizio 1. *Orari(codAula, nomeAula, piano, posti, materia, cdl, docente, giorno, oraInizio, oraFine)*. Il committente espone le seguenti regole:

1. In un dato momento, un docente si trova al più in un aula;
2. Non è possibile che due docenti diversi siano nella stessa aula contemporaneamente;
3. Se due lezioni si svolgono su due piani diversi appartengono a due corsi di laurea diversi;
4. Se due lezioni diverse si svolgono lo stesso giorno per la stessa materia, appartengono a due CDL diversi.

Le dipendenze funzionali sono:

1. Fissato il giorno, l'ora inizio, l'ora fine, il docente, è impossibile che quest'ultimo si trovi in due aule: `docente, giorno, oraInizio, oraFine` $\Rightarrow$ `codAula`. Anche `docente, giorno, oraInizio` $\Rightarrow$ `codAula` e `docente, giorno, oraFine` $\Rightarrow$ `codAula` sono dipendenze funzionali perchè è impossibile che è un professore è in aula 1 dalle 9 alle 11 e nello stesso momento dalle 10 alle 11. La prima dipendenza funzionale è derivabile dalle altre due, quindi si omette perchè indebolisce il vincolo di dipendenza funzionale;
2. Letteralmente sarebbe $\text{docente}_{\neq}$, $\text{giorno}_{=}$, $\text{oraInizio}_{=}$, $\text{coAula}_{=} \Rightarrow \bot$, ovvero, `giorno, oraInizio, codAula` $\Rightarrow$ `docente` oppure `giorno, oraFine, codAula` $\Rightarrow$ `docente`;
3. $\text{Piano}_{\neq} \Rightarrow \text{cdl}_{\neq}$, quindi ribaltando `cdl` $\Rightarrow$ `piano`;
4. Verrebbe da scrivere "lezione"$_{\neq}$, $\text{giorno}_{=}$, $\text{materia}_{=} \Rightarrow \text{cdl}_{\neq}$ dove lezione è una abbreviazione per tutti gli altri attributi, e quindi `giorno, materia, cdl` $\Rightarrow$ `"lezione"`. Il concetto di lezione diversa non si è aggiunta, per esempio, nel punto 2. perchè vi è gia il docente diverso, e quind sicuramente anche la lezione è diversa, cioè sarebbe stato ridondante aggiungerlo;

Terminologia sulle dipendenze funzionali:

- R <T, F> denota uno schema con attributi T e dipendenze funzionali F;

- Una dipendenza funzionale si dice **completa** quando X $\rightarrow$ Y è vera, ma se si considera un sottoinsieme stretto W di X allora non sarebbe più vera. Per esempio `cognome, matricola` $\rightarrow$ `nome` questo fatto è vero, ma non è espresso in maniera sintetica perchè se si togliesse "cognome" la dipendenza continuerebbe ad essere vera, quindi non è completa perchè c'è un attributo ridondante;
- Un insieme di attributi è **superchiave** quando è vera la dipendenza funzionale X $\rightarrow$ T, cioè X determina tutti gli altri attributi. Per esempio la coppia cognome, matricola nella tabella degli studenti è superchiave;
- **Chiave** è una superchiave minimale in cui non si può togliere nessun attributo senza perdere la proprietà di determinare tutti gli altri attributi. In altre parole se X è una chiave allora X $\rightarrow$ T è una dipendenza funzionale completa. Si dice che è un attributo è **primo** se appartiene ad almeno una chiave.

Esercizio 2. *Proiezioni(codiceCinema, nomeCinema, numeroSala, cittàCinema, data, oraInizio, codiceFilm, titoloFilm, registaFilm, prezzoBiglietto)*. Ogni ennupla rappresenta una proiezione di un film in un sala e un cinema ha in generale più sale, nelle quali avvengono le proiezioni. Si specifichi in quali delle seguenti regole si possono ricavare dipendenze funzionali:

1. CodiceCinema identifica il cinema;
2. Ogni film (identificato da un codiceFilm) ha un unico titolo e regista;
3. Due film diversi proiettati lo stesso giorno nella stessa città hanno titoli diversi;
4. Due sale diverse in cinema diversi possono avere lo stesso numeroSala;
5. Nessun regista fa due film con lo stesso titolo;
6. Quando uno stesso film è proiettato in più sale dello stesso cinema nello stesso giorno, le proiezioni iniziano ad ore diverse;
7. I prezzi dei biglietti sono uguali per tutte le proiezioni che avvengono in un cinema fissato un giorno fissato;
8. Non è possibile che lo stesso film sia proiettato lo stesso giorno in due cinema con prezzi diversi.

Le dipendenze funzionali sono:

1. `codiceCinema` $\Rightarrow$ `nomeCinema, cittàCinema` questo perchè ogni cinema risiede in un'unica città e nessuno degli altri attributi riguarda il cinema;
2. `codiceFilm` $\Rightarrow$ `titoloFilm, registaFilm`;
3. Sarebbe film$_{\neq}$, giorno$_{=}$, cittàCinema$_{=}$ $\Rightarrow$ titoloFilm$_{\neq}$ cioè `titoloFilm, data, cittàCinema` $\Rightarrow$ `codiceFilm`;
4. Verrebbe da scrivere sala$_{\neq}$, codiceCinema$_{\neq}$ $\Rightarrow$ numeroSala$_{=}$ ma nella regola c'è scritto "possono" mentre l'implicazione significa "deve" quindi non si può esprimere una dipendenza funzionale;
5. RegistaFilm$_{=}$, film$_{\neq}$, titoloFilm$_{=}$ $\Rightarrow \perp$ allora `registaFilm, titoloFilm` $\Rightarrow$ `codiceFilm`;
6. CodiceFilm$_{=}$, numeroSale$_{\neq}$, codiceCinema$_{=}$, data$_{=}$ $\Rightarrow$ oraInizio$_{\neq}$ cioè `codiceFilm, codiceCinema, data oraInizio` $\Rightarrow$ `numeroSale`;
7. `codiceCinema, data` $\Rightarrow$ `prezzoBiglietto`;
8. CodiceFilm$_{=}$, data$_{=}$, codiceCinema$_{\neq}$, prezzoBiglietto$_{\neq}$ $\Rightarrow \perp$, se si traducesse in questo modo ci sarebbe il problema di avere a destra due attributi: codiceCinema$_{=}$ OR prezzoBiglietto$_{=}$ e quindi non sarebbe più una dipendenza funzionale perchè per esserlo tutti gli attributi, a destra e a sinistra dell'implicazione, devono essere in AND. E' più facile interpretare questa frase immaginando che "in due cinema" sia ridondante, cioè non è indicato "due cinema diversi" o "due cinema uguali" ma lo si è interpretato in questo modo. Quindi rimuovendo codiceCinema diventa una dipendenza funzionale: `codiceFilm, data` $\Rightarrow$ `prezzoBiglietto`.

7.4 Regole di inferenza

Il problema dell'inferenza è stabilire se esistono un insieme di regole di ragionamento tali che applicando ripetutamente queste regole viene garantito tutto quello che si deve dimostrare. Un sistema di regole di inferenza che ha queste proprietà si chiama **Armstrong** ed è composto da 3 assiomi:

- **Riflessività R**, $\varnothing \vdash X \to Y$ se $Y \subseteq X$, dove $\vdash$ può essere tradotto con "si può dedurre meccanicamente questa cosa da quest'altra";
- **Arricchimento A**, $X \to Y \vdash XZ \to YZ$;
- **Transitività T**, $X \to Y, Y \to Z \vdash X \to Z$.

Dato un insieme di dipendenze funzionali F si può derivare $X \to Y$ ($F \vdash X \to Y$) se esiste una sequenza di passi di derivazione che parte dalle dipendenze funzionali di F e applicando gli assiomi di Armstrong arriva alla dipendenza $X \to Y$. Questa sequenza di applicazioni degli assiomi di Armstrong prende il nome di **derivazione** .

Esercizio 3. R $<$(ABCD), F = $\{A \to B, BC \to D\}>$, AC è una superchiave? Ovvero AC $\to$ ABCD?

1. $A \to B$ $\{$*Ipotesi 1*$\}$
2. $AC \to BC$ $\{$*Da 1 per* ***Arr(C)***$\}$
3. $BC \to D$ $\{$*Ipotesi 2*$\}$
4. $BC \to BCD$ $\{$*Da 3 per* ***Arr(BC)***$\}$
5. $AC \to BCD$ $\{$*Da 2+4 per* ***Trans****)*$\}$
6. $AC \to ABCD$ $\{$*Da 5 per* ***Arr(A)***$\}$

7.5 Chiusura di un insieme F

La **chiusura di un insieme F di dipendenze funzionali**, denotata con F^+, sono tutte quelle dipendenze $X \to Y$ che si possono ricavare da F, cioè $F^+ = \{X \to Y \mid F \vdash X \to Y\}$. Per esempio $F = \{A \to B\}$ e gli attributi disponibili sono (AB) allora $F^+ = \{A \to A, AB \to AB, AB \to A, AB \to B, B \to B, A \to B\}$. La chiusura di un insieme di dipendenze funzionali ha sempre dimensioni 2^N. La **chiusura di un insieme di attributi X** rispetto a F sono tutti gli attributi di T tali che X determina quei attributi, cioè $X_F{}^+ = \{A_i \in T \mid F \vdash X \to A_i\}$. Per esempio in uno schema $<$ABCDE$>$ con dipendenze funzionali $\{A \to B, B \to C\}$ allora $A_F{}^+ = ABC$, $B_F{}^+ = BC$, $C_F{}^+ = C$, $D_F{}^+ = D$, $E_F{}^+ = E$ oppure con queste dipendenze funzionali $\{AB \to C, C \to D\}$ allora $A_F{}^+ = A$, $AB_F{}^+ = ABCD$.

7.5.1 Algoritmo della chiusura lenta

L'algoritmo della chiusura lenta serve per calcolare la chiusura di un insieme di attributi. Ad ogni problema del tipo $F \vdash X \to Y$ per capire che è effettivamente vero basta calcolare la chiusura di X rispetto a + e controllare se contiene Y, cioè $Y \subseteq X_F{}^+$.

Esercizio 4. F = $\{DB \to E, B \to C, A \to B\}$, trovare AD^+

1. $AD^+ = AD$
   si inizializza $AD^+ = AD$ perchè AD è sempre derivabile a partire da AD. A questo punto si incomincia a scandire tutte le dipendenze per trovarne una che soddisfi la condizione, $A \subseteq AD^+$, perchè questo significa $AD^+ \to B$ per transitività, allora si aggiunge B agli attributi determinabili da X^+. Quando si trova una dipendenza che soddisfa la condizione, la si chiama **utilizzabile**
2. $AD^+ = ADB$
   siccome ora vi è un attributo in più, può darsi che qualche dipendenza che prima non era utilizzabile, ora lo sia, infatti, si può usare la prima dipendenza
3. $AD^+ = ADBE$
4. $AD^+ = ADBEC$

Quindi AD è una superchiave perchè determina tutti gli altri attributi.

7.6 Trovare una chiave in una relazione

In un relazione R con attributi (ABCDE) e con un insieme di dipendenze funzionali F, per trovare una chiave si parte da ABCDE e in seguito ci si chiede *l'attributo A lo si può togliere? ovvero BCDE $\rightarrow$ T?* dove T è l'insieme di tutti gli attributi. Per rispondere alla domanda, ci si pone un'altra domanda $T \in BCDE^{+}$*?* quindi si calcola la chiusura di BCDE, se A è di nuovo presente allora lo si poteva cancellare, altrimenti A è indispensabile. Si cancellano tutti gli attributi che si possono rimuovere fino a trovare una chiave.

7.7 Copertura di insiemi di DF

Due insiemi di dipendenze funzionali F e G, definite sullo stesso schema, sono equivalenti quando hanno la stessa chiusura, che si può esprimere come $F \subseteq G^{+}$ e $G \subseteq F^{+}$. Una dipendenza contiene attributi **estranei** quando non è completa, ovvero data una dipendenza `codice fiscale, matricola` $\Rightarrow$ `nome` si dice che, per esempio, matricola è estranea in questa dipendenza quando a partire da tutte le dipendenze che si hanno a disposizione, si può dedurre che il solo attributo codice fiscale determina il nome. Una dipendenza può essere **ridondante**, per esempio:

1. `codice fiscale` $\Rightarrow$ `matricola`
2. `codice fiscale` $\Rightarrow$ `nome`
3. `matricola` $\Rightarrow$ `nome`

la seconda dipendenza è ridondante perchè si può ricavare dalla uno + la terza per transitività. Se si rimuove una dipendenza ridondante, la chiusura non cambia. Una copertura è **canonica** se:

- tutte le dipendenze sono espresse in forma elementare, cioè la parte destra di ogni dipendenze funzionale è un attributo. Per esempio $X \rightarrow ABC$, si spezza in $X \rightarrow A$, $X \rightarrow B$, $X \rightarrow C$;
- non esistono attributi estranei;
- nessuna dipendenza è ridondante.

Esercizio 5. Si consideri R(AB $\rightarrow$ D, AD $\rightarrow$ B, A $\rightarrow$ C, C $\rightarrow$ A, BEC $\rightarrow$ D, ABCDE)

- *Ci sono attributi estranei in alcune dipendenze?*
  Per controllare, per esempio, se A in AB $\rightarrow$ D è estraneo, si deve verificare se B determina D da solo, per fare ciò si calcola B^{+} e se D appartiene alla chiusura allora A è estraneo. Per calcolare la chiusura di B si esaminano tutti gli elementi e si nota che nessuno degli elementi è incluso in D, quindi A non è estraneo. Ora si esamina A^{+} che è uguale ad AC allora nemmeno B è estraneo. Con AD $\rightarrow$ B si arriva a D^{+} = D e, come prima, A^{+} = AC, ovvero entrambi gli attributi non sono estranei. In A $\rightarrow$ C e C $\rightarrow$ A non vi sono attributi che possono essere estranei. L'ultima dipendenze BEC $\rightarrow$ D: B non è estraneo perchè EC^{+} = ECA, C non è estraneo perchè BE^{+} = BE, mentre BC^{+} = BCAD e quindi E è estraneo.
- *Ci sono dipendenze ridondanti, ovvero che possono essere ricavate da altre?*
  Per verificare se AB $\rightarrow$ D è ridondante basta calcolare $AB^{+}{}_{F - AB \rightarrow D}$ e verificare se D è contenuto. $AB^{+}{}_{F - AB \rightarrow D}$ = ABC e quindi non è ridondante rispetto a questo insieme di dipendenze. Stessa sorte per le altre dipendenze tranne $BEC^{+}{}_{F - BEC \rightarrow D}$ che è uguale ad BECAD e quindi ridondante.

  Se si portasse in forma canonica, quindi eliminando E, si troverebbe che $AB^{+}{}_{F - AB \rightarrow D}$ e $BC^{+}{}_{F - BC \rightarrow D}$ sono ridondanti perchè entrambi sono uguali ad ABCD.
- *Quante chiavi si riescono a trovare?*
  Se si cancellasse la E, o l'intera dipendenza BEC $\rightarrow$ D, si troverebbe lo stesso insieme di chiavi. Si parte da ABCDE, che è sicuramente una chiave, si calcola $BCDE^{+}$ e si vede che A è contenuto nuovamente allora si può eliminare. Si va avanti con CDE^{+} = CDEAB, quindi la B si può eliminare. La C e la D non si possono eliminare perchè rispettivamente DE^{+} = DE e CE^{+} = CEA. Infine CD^{+} = CDAB, non si riesce a raggiungere in nessun modo la E (anche perchè non è mai a destra di nessuna dipendenza) quindi CDE è una chiave. Si può cercare a trovare un'altra chiave cambiando l'ordine, per esempio EDCBA, e dopo una serie di calcoli si trova EBA. Per trovare tutte le chiavi, si devono provare tutte le permutazioni.

7.8 Decomposizione di schemi

Una decomposizione di uno schema è un insieme di schemi che contengono tutti i suoi attributi. Per esempio *Esami(matricola, nome, materia)* si può scomporre in:

1. *Studenti(matricola, nome)* e *Esami(matricola, materia)*;
2. Studenti(matricola, nome) e *Esami(materia)*;
3. *Studenti(matricola, nome)* e *Esami(nome, materia)*.

Si chiama decomposizione perchè non si è perso nessun attributo. La decomposizione è utile perchè nella tabella *Esami(matricola, nome, materia)* è presente molta ridondanza, quindi la si vuole eliminare ma senza perdere la conservazione dei dati e delle dipendenze. Una decomposizione di una tabella R(T) preserva i dati quando per qualunque sia l'istanza valida r della tabella si proiettano su T_1, ..., T_k, poi si effettua tutta la giunzione, e si trovano solo le ennuple di partenza, cioè se **per ogni istanza valida r di R vale r = (π_{T_1} r) $\bowtie$ (π_{T_2} r) $\bowtie$ ... $\bowtie$ (π_{T_k} r)**. Quando la decomposizione è **binaria**, cioè è composta solo da due tabelle T_1 e T_2, allora preserva i dati se e solo se $T_1 \cap T_2 \to T_1 \in F^+$ oppure $T_1 \cap T_2 \to T_2 \in F^+$, cioè se l'intersezione delle due tabelle è chiave per T_1 o T_2. Nell'esempio 1, l'intersezione delle due tabelle è matricola, e matricola è chiave per gli Studenti ma non per gli Esami, quindi questa intersezione determina la tabella degli Studenti allora questa decomposizione è corretta. Nell'esempio 2, l'intersezione è l'insieme vuoto e quindi la decomposizione non è corretta. Nell'esempio 3, l'intersezione è l'attributo nome, e quest'ultimo non è chiave per gli studenti, allora la decomposizione non è corretta. **Teorema:** Se si ha una decomposizione che preserva le dipendenze e almeno uno dei sottoschemi è una superchiave dello schema di partenza, allora questa decomposizione preserva i dati.

7.8.1 Proiezione delle dipendenze

In generale se uno schema R viene decomposto in insiemi T_1, T_2, ..., T_n alcune delle dipendenze dello schema di partenza non si riescono più ad esprimere. Per esempio: sia R(A, B, C) e $F = \{A \to B, B \to C, C \to A\}$, e siano le proiezioni:

- $\pi_{AB}(F) \equiv \{A \to B, B \to A\}$
- $\pi_{AB}(F) \equiv \{A \to C, C \to A\}$

La dipendenza $B \to C$ non si riesce più a esprimere perchè nella prima manca l'attributo C, mentre nella seconda manca l'attributo B. Dato uno schema e una decomposizione, lo schema preserva le dipendenze se l'unione di tutte le dipendenze che si trovano in tutte le decomposizioni compre l'F iniziale. Nell'esempio di prima, l'unione di tutte le dipendenze di tutte le decomposizioni è $\{A \to B, B \to A, A \to C, C \to A\}$ e questo insieme copre F perchè $A \to B$ è derivabile, anche $C \to A$ e per $B \to C$ si calcola $B^+ = BAC$, quindi C è contenuto, per cui è derivabile. **Teorema**: se una decomposizione preserva le dipendenze e almeno uno dei sottoschemi è una superchiave per lo schema di partenza allora questa decomposizione preserva i dati.

7.9 Forme normali

Vi sono diverse nozioni di forme normali:

- **1FN**, ogni attributo di una relazione ha un tipo elementare;
- **2FN, 3FN, FNBC**, sono forme più restrittive che eliminano le ridondanze. FNBC è la più naturale e la più restrittiva.

FBNC: dato uno schema R <T, F> è in BCNF se e solo se per ogni dipendenza funzionale $X \to A \in F^+$, con $A \notin X$ (non banale), X è una superchiave. Detto in altri termini se $A \notin X$ allora X è una superchiave. Teorema: dato uno schema R <T, F> è in BCNF se e solo se per ogni dipendenza funzionale $X \to A \in F$, $A \notin X$ o X è una superchiave. Esempi:

- *Docenti(<u>codiceFiscale</u>, nome, dipartimento, indirizzo)*
  se si considera `dipartimento` $\Rightarrow$ `indirizzo`, questa dipendenza non è in BCNF perchè dipartimento non è una superchiave, infatti questa tabella ha delle ridondanze perchè si può trovare, per esempio, tre volte il dipartimento di informatica e si troverà tre volte con lo stesso indirizzo;

- *Impiegati(codice, qualifica, nomeFiglio)*
  quando l'impiegato ha tre figli, si dedicano tre righe, per ciascun figlio, all'impiegato. Non vi è in BCNF perchè il codice determina la qualifica;
- *Librerie(codiceLibro, nomeNegozio, indNegozio, titolo, quantità)*
  `codiceLibro ⇒ titolo`, `nomeNegozio ⇒ indNegozio` e `codiceLibro, nomeNegozio ⇒ quantità` quindi codiceLibro e nomeNegozio insieme sono chiave, ma singolarmente nomeNegozio non è chiave per cui non dovrebbe determinare indNegozio, se ciò accade significa che è una ridondanza perchè tutte le volte che si ripete il nome del negozio, si deve ripetere anche lo stesso indirizzo;
- *Telefoni(prefisso, numero, località, abbonato, via)*
  `prefisso numero ⇒ località, abbonato, via` e `località ⇒ prefisso` ma località non è chiave perchè non determina il numero, anche questo schema non è in BCNF.

7.9.1 Algoritmo di analisi

L'algoritmo di analisi trasforma un qualunque schema in BCNF, preservando i dati ma non sempre le dipendenze:

- *Docenti(codiceFiscale CF, nome N, dipartimento D, indirizzo I)* avente come dipendenze funzionali `codiceFiscale ⇒ nome, dipartimento` e `dipartimento ⇒ indirizzo`
  l'algoritmo inizia esaminando le dipendenze funzionali, per vedere se sono in BCNF: codiceFiscale rispetta la BCNF? ha a sinistra una superchiave? si calcola la chiusura del codice fiscale CF^+ = CF, N, D, I, quindi CF è chiave perchè cattura tutti gli attributi e allora non viola la BCNF. Si esamina la seconda dipendenza: si calcola D^+ = D, I, quindi D non è chiave per cui la seconda dipendenza viola la BCNF. Allora si decompone R in:
  - R1 = {dipartimento, indirizzo}
  - R2 = {codiceFiscale, nome}

  Con questa decomposizione non si preservano i dati perchè queste due tabelle R1 e R2 non hanno nulla in comune, per preservare i dati si deve fare in modo che vi sia una intersezione che è chiave per una delle due tabelle:
  - R1 = {dipartimento, indirizzo}
  - R2 = {codiceFiscale, nome, dipartimento*}

  dove dipartimento in R2 fa da chiave esterna. A questo punto si fa la proiezione delle dipendenze funzionali: per R1 `dipartimento ⇒ indirizzo` ora questa dipendenza è in forma BCNF, per R2 `codiceFiscale ⇒ nome, dipartimento` ma era già in forma BCNF, quindi l'algoritmo termina preservando dati e dipendenze.
- *Impiegati(codice C, qualifica Q, nomeFiglio NF)* con dipendenza `codice ⇒ qualifica`
  si esamina la dipendenza, C^+ = CQ che non è chiave quindi si deve decomporre ma aggiungendo ad una tabella il determinante della dipendenza:
  - R1 = {codice, qualifica}
  - R2 = {nomeFiglio, codice*}

  Si proiettano le dipendenze dove per R1 `codice ⇒ qualifica` che adesso è in forma BCNF. R2 non ha nessuna dipendenza. Questa decomposizione preserva i dati e le dipendenze.
- *Telefoni(prefisso P, numero N, località L, abbonato A, via V)* con dipendenze `prefisso numero ⇒ località, abbonato, via` e `località ⇒ prefisso`
  si esaminano le dipendenze PN^+ = PNLAV quindi rispetta BNCF, mentre L^+ = LP quindi non rispetta BNCF. La decomposizione sarebbe:
  - R1 = {località, prefisso}
  - R2 = {numero, abbonato, via, località*}

  Si proiettano le dipendenze dove per R1 `località ⇒ prefisso` rispetta la forma BCNF, mentre R2 non ha nessuna dipendenza perchè si è perso la dipendenza `prefisso numero ⇒ località, abbonato, via`. Dato che la chiusura approssimata ha perduto una dipendenza, si deve calcolare la chiusura esatta: si devono andare a verificare tutti i sottoinsiemi di questi quattro attributi e vedere quali non hanno una chiusura non banale. Dopo un calcolo un po' laborioso (...) si arriva a `località numero ⇒ abbonato, via`. A questo punto si è in forma BCNF.

- *R(ABCDE, $\{AB \to D, C \to B, D \to E\}$)*
  AB $\to$ D, non è in forma BCNF perchè AB^+ = ABDE, quindi si scompone e si proiettano le dipendenze:
  - R1 = $\{$ABDE$\}$, AB $\to$ D, D $\to$ E
  - R2 = $\{$CAB$\}$, C $\to$ B

  D $\to$ E non è in forma BNCF perchè D non è chiave quindi R1 va decomposta a sua volta:
  - R3 = $\{$DE$\}$, D $\to$ E
  - R4 = $\{$ABD$\}$ AB $\to$ D

  Ora è in forma BCNF. Anche C $\to$ B non è in forma BCNF:
  - R5 = $\{$CB$\}$, C $\to$ B
  - R4 = $\{$CA$\}$ nessuna dipendenza

  Ora tutte le tabelle sono in forma BCNF e non si è perso nessuna dipendenza.
- *R(ABCDE, $\{AB \to D, AC \to B, D \to C\}$)*
  AB^+ = ABDC quindi non è in forma BCNF, quindi si scompone e si proiettano le dipendenze:
  - R1 = $\{$ABDC$\}$, AB $\to$ D, AC $\to$ B, D $\to$ C
  - R2 = $\{$EAB$\}$, nessuna dipendenza

  si effettua una nuova analisi: D $\to$ C non è in forma BCNF perchè D^+ = DC quindi va decomposto un'altra volta:
  - R3 = $\{$DC$\}$, D $\to$ C
  - R4 = $\{$ABD$\}$, AB $\to$ D

  AC $\to$ D è perduto, a questo punto si deve calcolare la proiezione esatta delle dipendenze: si devono verificare tutti i sottoinsiemi di ABD:
  - A^+ = A
  - B^+ = B
  - D^+ = D
  - AB^+ = ABD
  - AD^+ = ADCB, quindi vi è una dipendenza derivata AD $\to$ B perchè B è presente in R4
  - BD^+ = BD

  Quindi la proiezione completa di R4 è AB $\to$ D e AD $\to$ B. I sottoinsiemi di DC:
  - D^+ = DC
  - C^+ = C

  A questo punto lo schema è in forma BCNF.

7.9.2 Terza forma normale

La Forma Normale di Boyce Codd ha il problema che in alcuni schemi non ammette nessuna decomposizione che sia contemporaneamente in FNBC e che preservi dati e dipendenze. Per risolvere il problema è stato definito un'altra nozione di forma normale: **la terza forma normale**. Uno schema è in 3FN se e solo se per ogni dipendenza X $\to$ A $\in F^+$, quando A $\notin$ X allora X è una superchiave o A è primo. La terza forma normale è più permissiva, cioè accetta un maggior numero di schemi. Se uno schema non è in 3FN allora sicuramente non è nemmeno in BCNF. Teorema: Uno schema è in 3FN se e solo se per ogni dipendenza X $\to$ A $\in$ F, quando A $\notin$ X allora X è una superchiave o A è primo. Esempi:

- *Docenti(codiceFiscale, nome, dipartimento, indirizzo)* avente come dipendenze funzionali `codiceFiscale` $\Rightarrow$ `nome, dipartimento` e `dipartimento` $\Rightarrow$ `indirizzo`
  non è in 3FN perchè nella seconda dipendenza l'indirizzo non fa parte di nessuna chiave;
- *Impiegati(codice, qualifica, nomeFiglio)* con dipendenza `codice` $\Rightarrow$ `qualifica`
  non è in 3FN perchè l'unica chiave è la coppia codice - nomeFiglio e la qualifica non ne fa parte;

- *Telefoni(prefisso, numero, località, abbonato, via)* con dipendenze `prefisso numero ⇒ località, abbonato, via` e `località ⇒ prefisso`
  le chiavi sono le coppia prefisso - numero e località - abbonato, ed è in 3FN perchè il prefisso fa parte della chiave;

- *Esami(matricola, telefono, materia, voto)* con dipendenze `matricola materia ⇒ voto`, `matricola ⇒ telefono`, `telefono ⇒ matricola`
  le chiavi sono le coppi matricola - materia e telefono - materia, è in forma 3FN perchè sia telefono che matricola fanno parte di una chiave

7.9.3 Algoritmo di sintesi

L'algoritmo di sintesi trasforma qualunque schema in 3FN preservando dati e dipendenze, ma non garantisce che lo schema sia sempre anche in BCNF. L'algoritmo inizia partendo da una copertura canonica, dopodichè si partiziona questo insieme di dipendenze in gruppi in base al determinante. Per esempio se ci fossero le dipendenze: $\{AB \to C$, $A \to D$, $AB \to E$, $B \to C\}$ si divide in 3 gruppi:

- $AB \to C$ e $AB \to E$
- $A \to D$
- $B \to C$

cioè ogni determinante diverso dà luogo a un gruppo diverso. Dopo la partizione si definisce uno schema di relazione per ciascun gruppo, nell'esempio per il primo gruppo si fa corrispondere lo schema R1(ABCE), al secondo gruppo lo schema R2(AD), e al terzo gruppo lo schema R3(BC). Si faccia attenzione che in ogni gruppo non fa fatto la chiusura, per esempio di AB, ma semplicemente è l'unione di tutti gli attributi che appaiono in quelle dipendenze. Il prossimo passo dell'algoritmo è verificare che uno degli schemi è già incluso in altri. Nell'esempio R3(BC) può essere eliminato perchè è già incluso in R1. Come l'ultimo passaggio si verifica se almeno uno degli schemi creati è superchiave per lo schema di partenza: si calcola $ABCE^+$ = ABCED quindi R1 è già superchiave per la tabella di partenza. Se nella tabella di partenza ci fosse stato anche un attributo F che non appare in nessuna dipendenza funzionale allora l'algoritmo dice di aggiungere un ulteriore tabella che contenga una chiave scelta arbitrariamente: R4(ABF). Si studiano altri esempi:

- *Telefoni(prefisso, numero, località, abbonato, via)* con dipendenze `prefisso numero ⇒ località, abbonato, via` e `località ⇒ prefisso`
  l'algoritmo di sintesi costruisce due tabelle:

  1. `prefisso numero ⇒ località, abbonato, via`
  2. `località ⇒ prefisso`

  la seconda dipendenza viene rimossa perchè è incluso nella prima e dato che la prima dipendenza è superchiave l'algoritmo di sintesi restituisce in output esattamente lo stesso schema di partenza;

- *Esami(matricola, telefono, materia, voto)* con dipendenze `matricola materia ⇒ voto`, `matricola ⇒ telefono`, `telefono ⇒ matricola`
  l'algoritmo di sintesi produce tre schemi:

  1. `matricola materia ⇒ voto`
  2. `matricola ⇒ telefono`
  3. `telefono ⇒ matricola`

  la terza dipendenza è contenuta nella seconda dipendenza e la prima dipendenza è superchiave. Si è generato una decomposizione che preserva i dati e le dipendenze, ed è in 3FN e BCNF

Esercizio 6. Si consideri lo schema: $R<(ABCDE), \{CE \to A, CD \to E, CD \to A, B \to A, CE \to D)\} >$

1. Trovare se ci sono dipendenze ridondanti. Se sì, cancellarle prima di passare alle domande successive;
2. Trovare una chiave;
3. Eseguire l'algoritmo di *sintesi* sullo schema originario senza dipendenze ridondanti;
4. Eseguire l'algoritmo di *analisi* sullo schema originario senza dipendenze ridondanti

Risposte:

1. Per vedere se CE è ridondante si calcola la chiusura di CE rispetto alle altre dipendenze: CE^+ = CEDA, include già A quindi CE è ridondante. Cancellandola si arriva ad un insieme senza ridondanze. Il nuovo schema è il seguente: R<(ABCDE), $\{CD \to E, CD \to A, B \to A, CE \to D)\}$ >

2. ABCDE è una superchiave, A si può eliminare perchè è di nuovo incluso con la seconda e terza dipendenza; B non si può eliminare; Stesso discorso per C; D si può eliminare per l'ultima dipendenza; BC^+ = BCA quindi la E è indispensabile. Quindi BCE è una chiave;

3. Si raggruppano le dipendenze per determinante

   - $CD \to E$ e $CD \to A$
   - $B \to A$
   - $CE \to D$

   Si determinano i gruppi: R1(CDEA), R2(BA) e R3(CED), dove R3 è incluso in R1 quindi lo si può eliminare. Si verificano se uno dei due gruppi rimanenti è chiave: entrambi non lo sono perchè R1 non determina la B e R2 non determina la C, quindi si aggiunge un terzo schema R4(BCE). Questa è una decomposizione che preserva dati e dipendenze

4. $CD \to E$ non è in forma BCNF perchè CD^+ = CDEA quindi si scompone e si proiettano le dipendenze:

   - R1 = $\{CDEA\}$, $CD \to E$, $CD \to A$, $CE \to D$
     $B \to A$ è perduto quindi si deve calcolare la proiezioni esatta:

     - A^+ = A
     - C^+ = C
     - D^+ = D
     - E^+ = E
     - AC^+ = AC
     - AD^+ = AD
     - AE^+ = AE
     - CD^+ = CDEA
     - CE^+ = CDEA

     si dovrebbe continuare a controllare le altre dipendenze ma si nota che l'unico modo per scoprire qualche dipendenza nuova è avere un calcolo con $B \to A$, ma purtroppo non si riuscirà mai ad utilizzarla perchè B non è a destra di nessuna altra dipendenza. In R1 non ci sta nient'altro da scoprire
   - R2 = $\{CDB\}$, $CD \to E$, $CD \to A$
     $B \to A$ è perduto, proiezione esatta:

     - B^+ = BA
     - C^+ = C
     - D^+ = D
     - BC^+ = BCA
     - BD^+ = BDA
     - CD^+ = CDAE

     quindi nemmeno in R2 si riesce a scoprire nulla di nuovo

   L'algoritmo di analisi garantisce BCNF, mentre l'algoritmo di sintesi garantisce la preservazione delle dipendenze ma a volte, come in questo caso, l'algoritmo di sintesi restituisce anche BCNF e l'algorimo di analisi restituisce anche la preservazione delle dipendenze. Se si è interessati a ambedue gli obiettivi allora in linea teorica si dovrebbe provare entrambi gli algoritmi.

7.10 Dipendenze multivalore

Sono state elaborate anche altre nozioni di dipendenza funzionale e di forma normale per individuare altri tipi di anomalie, in particolare è stata sviluppata la teoria delle **dipendenze multivalore**. Le dipendenze multivalore specificano che esistono alcuni tipi di ridondanza che non si rappresentano con le dipendenze funzionali.

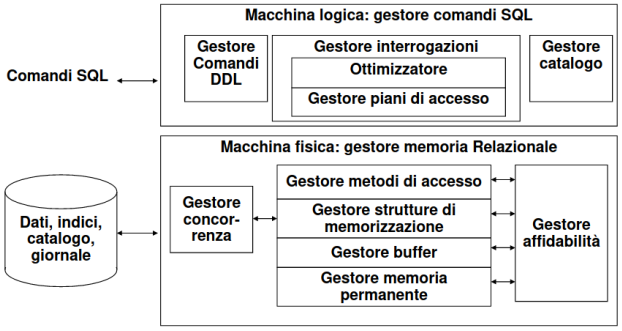
Chapter 8

Architettura dei DBMS

8.1 Architettura standard

I sistemi relazionali standard hanno una architettura divisa in due moduli:

- **Macchina fisica**, implementa le transazioni quindi atomicità, persistenza e serializzabilità, implementa l'accesso al disco, gli indici cioè tutti gli strumenti che velocizzano l'accesso al disco e mette a disposizione una interfaccia fisica che ha come operazioni fondamentali quelle di: *apri una tabella, scorri tabella, chiudi tabella, apri un indice, scorri indice, chiudi indice*;
- **Macchina logica**, è il gestore dell'SQL. È uno strumento che prende un comando SQL e lo traduce in un piano di accesso, ovvero una sequenza di operazioni del tipo: *apri questa tabella, scorri tabella, chiudi tabella, apri questo indice, scorri indice, chiudi indice*. L'ottimizzatore deve scegliere, tra tutti i possibili piani di accesso, quello idealmente ottimo

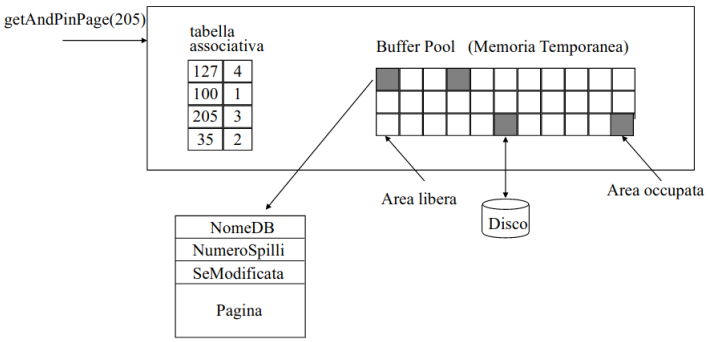


8.2 Macchina fisica

La macchina fisica contiene diversi moduli:

- **Gestore della memoria permanente**, che nasconde le differenze tra un sistema operativo e un altro;
- **Gestore del buffer**, che mette a disposizione una area di memoria centrale chiamata *buffer*. È quello strumento che si occupa del trasferimento delle pagine tra memoria permanente e la memoria temporanea, e si ricorda dove ha messo le pagine nel caso in cui esse fossero già state portate dal disco alla memoria temporanea. Il buffer contiene un certo insieme di pagine chiamato *buffer pool* che risiede nella RAM. Ciascuna di queste pagine, in ogni momento, è allocata ad una pagina sul disco, e vi è a questo scopo una tabella associativa bidirezionale che indica per ogni pagina sul disco attualmente allocata qual è la pagina del buffer che li corrisponde e viceversa. Quando un processo ha bisogno di accedere a una pagina sul disco, il processo manda al buffer un comando del tipo *getPage*, per esempio getPage(205), e il buffer fa la seguente cosa: cerca nella tabella associativa, e se è già presente la pagina 205 ed è in posizione 3 allora il gestore del buffer risponde 3 e incrementa di uno sulla pagina 3 il numero degli utenti (numeroSpilli) che stanno usando quella pagina. Quando il processo non avrà più bisogno di quella pagina, darà un comando *releasePage(205)* e si decrementa di uno il numero degli utenti per quella pagina. Quando il processo *getPage()* va a modificare la pagina allora il processo deve mettere a 1 il bit *seModificata*. Il numero degli utenti e il bit seModificata sono fondamentali perchè il buffer a forza di rispondere a *getPage()* a un certo punto avrà tutte le pagine usate, quindi alla prossima richiesta di *getPage()* il gestore del buffer deve scegliere una pagina da rimpiazzare. Una pagina per essere riciclata ha bisogno che il numero degli utenti sia zero e se il bit modifica è uguale a 1, non si può rilasciare la pagina ma prima va salvata sul disco. Questa operazione si chiama **flush**. Dopo aver scritto la pagina sul disco, il bit seModificata può essere messo a zero. Il gestore del buffer utilizza l'algoritmo **LRU** che ricicla la pagina meno usata, e la **politica di scrittura** che può essere:

- *Ansiosa*, il gestore del buffer non appena vede il bit *seModificata* uguale a 1, ha fretta di trasformarlo in zero il primo possibile, cioè ogni scrittura viene subito salvato sul disco. Una politica ansiosa fa perdere tutti i vantaggi del buffer di scrittura;
- *Pigra* è la politica apposta a quella ansiosa, dove si va a flushare sul disco solo nel momento in cui vi è una necessità di rimpiazzamento. Questa politica fornisce la massima efficienza, il minor numero di operazioni sul disco ma che può avere lo svantaggio come una grossa divergenza tra la memoria centrale e il disco. Questa divergenza può rallentare il restart dopo i fallimenti;
- Un'altra politica possibile, che si tratterà in seguito, è quella del *checkpoint*.



- **Gestore delle strutture di memorizzazione** e **gestore dei metodi di accesso** che mettono a disposizione dei meccanismi per implementare tabelle e indici;
- **Gestore affidabilità**, implementa atomicità e persistenza, ovvero fa si che la macchina sia affidabile anche in presenza di guasti;
- **Gestore concorrenza**, implementa la serializzabilità ovvero fa si che diverse transazioni in parallelo fra di loro non creino conflitti

8.2.1 Memorie persistenti

Le memorie persistenti si dividono in due categorie:

- **HDD**, è uno strumento meccanico dotato di un disco girevole e una testina che si muove a destra o a sinistra per andare a leggere le diverse tracce. I dati del disco vengono trasferiti in blocchi e mai un byte alla volta, questo per l'eccessiva lentezza del disco. I blocchi generalmente sono 4 Kbyte;
- **SSD**, è uno strumento che non contiene parti rotandi ma dei chip. Per accedere a una porzione di memoria si indica l'indirizzo logico al controller che lo traduce in un indirizzo fisico e va ad indirizzare la porzione di memoria che si voleva accedere. I dati sono acceduti per blocchi, come sul disco, ma per ragioni diverse. Nell'HDD il tempo di lettura è dominato dal tempo di latenza, cioè il tempo che si aspetta per cominciare a leggere. Nell'SSD non c'è da attendere un tempo di latenza quindi la velocità di accesso è molto più veloce rispetto ad un HDD. Gli SSD sono più affidabili perchè appunto non sono meccanici, ma sono molto più costosi e hanno una capacità minore. Infine come ultimi vantaggi gli SSD hanno consumi elettrici minori e una tecnologia in crescita.

8.3 Organizzazione dati su disco

Ogni file è diviso in una sequenza di pagine consecutive, e ogni pagina contiene un insieme di record. Per ogni record esiste un RID, cioè un puntatore al record, che è composto da una sequenza di bit di cui quelli più significativi identificano la pagina, mentre quelli meno significativi identificano una entry all'interno di un array che è memorizzato in ogni pagina. I riferimenti ai record sono composti dalla coppia: PID (ID della pagina) e RID. Si suppone che le operazioni all'interno della pagina hanno costo nullo, la unica operazione costosa è portare una pagina dal disco al buffer. L'unità di costo è l'operazione di input/output cioè quante letture e scritture si devono fare per inserire un record. I dati possono essere organizzati con vari algoritmi:

- **Organizzazione seriale**, chiamata anche heap, è l'organizzazione più banale dove ogni record viene inserito in fondo al file. I dati sono memorizzati in modo disordinato uno dopo l'altro. Questa organizzazione ha il minor costo di memoria ed è molto efficiente per l'inserimento perchè in generale basta tenere l'ultima pagina all'interno del buffer e compiere l'inserimento. Un'altra operazione efficiente è la scansione totale (*table scan*). È poco efficiente per le ricerche per valore e per intervallo perchè si deve scandire tutta la tabella, anche per le modifiche è inefficiente perchè per cambiare un dato, prima lo si deve trovare nella tabella. L'inserimento ha costo medio 1/D dove D è la densità, cioè il numero di record per pagine. Se nella pagina sono contenuti 20 record, l'unita di costo è 1/20 perchè non si deve leggere niente (poichè la pagina è già nel buffer) e si deve scrivere 1 volta ogni 20 record. Se l'inserimento viene effettuato in un file seriale usato molto di rado allora non si può più assumere che l'ultima pagina sia sul buffer ma si deve leggere l'ultima pagina (1), aggiornarla in memoria centrale aggiungendo il record (0) e poi scriverla sul disco (1) quindi ora il costo sarà 2. L'organizzazione seriale è l'organizzazione standard di ogni DBMS;

- **Organizzazione sequenziale**, i dati sono ordinati sul valore di uno o più attributi, e quando arriva un nuovo record, lo si inserisce nel punto giusto eventualmente spostando di uno tutti i record che si trovano dopo. Le ricerche sono piuttosto veloci, per la ricerca di un singolo elemento, si procede con un algoritmo di ricerca binaria (o di bisezione) e quindi avrà un costo $log_2\ NPagTab$. L'inserimento ha un costo molto elevato per cui l'organizzazione sequenziale è spesso ottimizzata con una tecnica di organizzazione sequenziale dinamica: ogni pagina non viene riempita completamente, ma si lascia un po' di spazio in fondo, in questo modo quando si deve inserire un nuovo record basta trovare la pagina in cui deve stare con un costo di $log_2\ NPagTab$ e poi lo si inserisce nello spazio vuoto con costo 1. Quando la pagina diventa completamente piena la si fonde con la pagina precedente. Se anche la pagina precedente è piena, allora da due pagine se ne ricava tre, tutte con un po' di spazio in fondo ma questo richiede che i file siano memorizzati in modo tale che sia possibile inserire una nuova pagina in mezzo al file e non fino in fondo;

- **Organizzazione per chiave**, esistono due famiglie di organizzazioni che ottimizzano l'accesso sulla base di un attributo e sono:

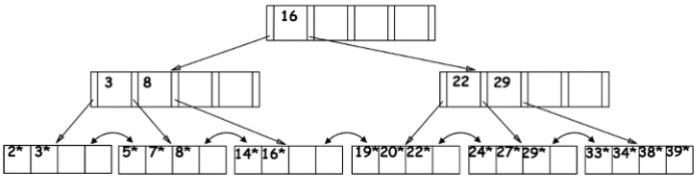
  – *Metodo procedurale (o hash)*, funziona in questo modo:

    1. si sceglie una funzione hash;
    2. si sceglie il fattore di caricamento cioè quanto si caricano le pagine (90%, 50%, ...);
    3. nel caso in cui ci sono molti conflitti su una pagina, si deve attuare un metodo per la gestione dei trabocchi. Si assume che ogni pagina abbia associata, nel caso in cui trabocca, una pagina con tutti gli elementi che sono traboccati.

    L'organizzazione hash è ottima per ricerca per uguaglianza, inserimento, aggiornamento e la cancellazione ma come punti di debolezza vi sono:

    * la staticità, più passa il tempo, più si aggiungono record e quindi l'organizzazione ha bisogno di una riorganizzazione, cioè ad un certo punto si deve ricostruire da zero il file ridimensionando le pagine su più pagine;
    * non supporta la ricerca per intervallo, l'unica possibilità è effettuare la table scan, cioè leggere tutte le pagine dell'organizzazione una dopo l'altra.

  – *Metodo tabellare (o ad albero)*, il metodo procedurale è utile per ricerche per uguaglianza ma non per intervallo. Quando si vuole ottimizzare entrambe le ricerche, si modellano i dati come un **B^+ - albero**: nelle foglie dell'albero i dati sono organizzati nel metodo sequenziale dinamico, cioè sono ordinati, ogni pagina contiene un po' di spazio libero e tutte le pagine sono collegate tra di loro con una lista doppia che permette di scandire i dati in avanti e in indietro, e permette in caso di overflow di inserire una nuova pagina in mezzo alla lista. Oltre al livello delle foglie, esiste un livello intermedio chiamato **indice sparso**, che funziona come un albero binario di ricerca. Ogni nodo corrisponde a una pagina, e ogni pagina contiene un certo numero di chiavi. In ogni nodo dell'albero si copia il valore più grande della chiave. Il B^+ albero è una organizzazione perfettamente dinamica.



Il B^+ albero è utilizzato spesso per memorizzare, non tanto i dati ma gli **indici**. Un indice è una struttura dati ausiliaria che organizza una tabella associativa tra un attributo e un RID. La tabella viene implementata come B^+ albero ma

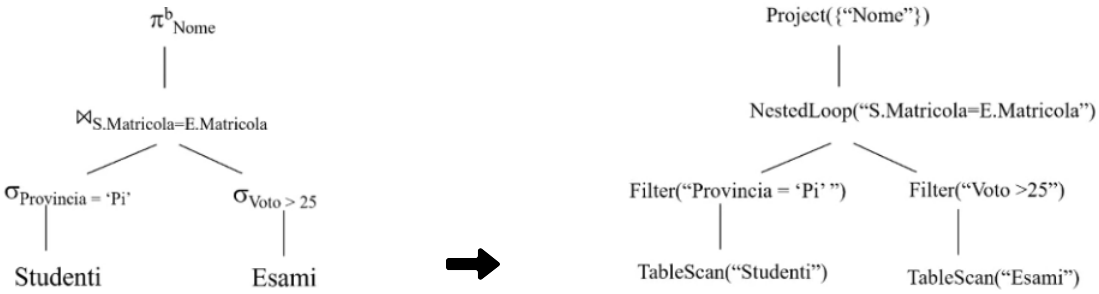
può essere organizzato anche secondo metodo hash. L'importante è che sia un organizzazione veloce, altrimenti non si ha nessun vantaggio nell'utilizzare un indice

8.4 Piani di accesso

Il piano di accesso è il modo in cui una interrogazione è effettivamente eseguita. Il sistema traduce l'interrogazione in SQL

```
SELECT Nome
FROM Studenti S, Esami E
WHERE S.matricola = E.matricola AND provincia = 'PI' AND voto > 25
```

in un albero logico, che poi viene tradotto a sua volta in un albero fisico.



L'albero fisico, viene chiamato **piano di accesso**, ed è una espressione algebrica in cui per ogni operatore logico, dell'albero logico, si specifica il nome dell'algoritmo che si userà per implementarlo. Ogni operatore fisico è realizzato con un oggetto con quattro metodi:

- **open()**, inizializza lo stato dell'operatore ed esegue il metodo open degli operatori fisici dei suoi argomenti;
- **next()**, ritorna il record successivo del risultato interagendo con gli operatori fisici dei suoi argomenti;
- **isDone()**, ritorna *true* se non ci sono altri record da ritornare, *false* altrimenti;
- **close()**, termina le operazioni dell'operatore fisico e dei suoi argomenti.

Per accedere ad una tabella esistono 3 operatori:

- **TableScan (R)**, per la scansione della tabella R;
- **IndexScan (R, Idx)**, scandisce la tabella R in ordine di indice Idx. L'indexScan è molto più lenta rispetto alla tableScan perchè accede ai record in tabella in modo disordinato, ma il risultato restituito è già ordinato;
- **SortScan (R,** $\{\mathbf{A_i}\}$**)**, ritorna la collezione dei record di R ordinati sui valori degli $\{A_i\}$;

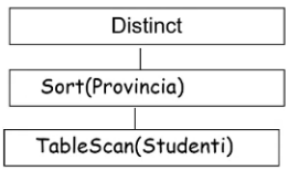
8.4.1 Operatori di proiezione

Per la proiezione dei record senza l'eliminazione dei duplicati si utilizza l'operatore **Project (O,** $\{\mathbf{A_i}\}$**)**, che ha come argomento la collezione dei record O restituiti da un atro operatore fisico. Per eliminare i duplicati vi è l'operatore **Distinct (O)**. Si consideri la query:

```
SELECT DISTINCT provincia
FROM Studenti R
```

l'algoritmo parte da qualche piano di accesso che restituisce tutti gli elementi che si vuole ordinare, come un *TableScan*, si ordinano gli elementi e infine si applica l'operatore *Distinct* che funziona in questo modo:

1. esegue next(), ritorna la ennupla che ha ricevuto e la memorizza;
2. al prossimo next(), *Distinct* va a leggere la prossima ennupla, la confronta con quella memorizzata e se è uguale ad essa non la restituisce, la scarta e chiede un altro next();
3. si va avanti con questo procedimento fino a che non arriva un **break**, cioè quando la prossima ennupla che arriva è diversa dall'ultima ennupla che il *Distinct* ha visto, e a questo punto il *Distinct* emette la ennupla e la memorizza.



La *Distinct* funziona solo se l'input che riceve è ordinato, perchè l'algoritmo funziona in modo che elimina i duplicati consecutivi.

8.4.2 Operatori per la restrizione dei dati

- **Filter (O, ψ)**, ritorna la collezione dei record di O che soddisfano la condizione ψ;
- **IndexFilter (R, Idx, ψ)**, ritorna la collezione dei record di R che soddisfano la condizione ψ, con l'uso dell'indice Idx definito su attributi di R;
- **Sort (O, $\{A_i\}$)**, per ordinare i record rispetto a determinati attributi.

8.4.3 Operatori per la giunzione

L'algoritmo di base è il **NestedLoop(O_E, O_I, ψ_j)** ritorna la giunzione dei record di O_E e O_I che soddisfano la condizione ψ_j:

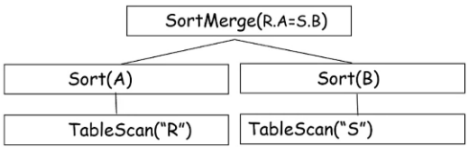
```
for each record r in R do
  for each record s in S do
      if r_i = s_j then
         aggiungi <r,s> al risultato
```

nella pratica non si utilizza mai perchè l'algoritmo ha un costo elevato. Un altro operatore è il **PageNestedLoop (O_E, O_I, ψ_j)** che è una ottimizzazione del *NestedLoop*. I due algoritmi che si utilizzano nella pratica sono:

- **IndexNestedLoop (O_E, O_I, ψ_j)**, ritorna la giunzione dei record di O_E e O_I che soddisfano la condizione di giunzione ψ_j, supponendo che esista un indice Idx sugli attributi di giunzione della relazione interna;

  l'*indexFilter* viene aperta tutte le volte con un valore diverso. Quando si scrive un piano di accesso con *indexNestedLoop* deve avere un *indexFilter* a destra e la condizione ψ_j deve essere la stessa;

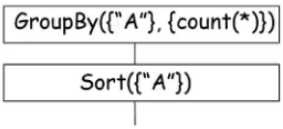
- **SortMerge (O_E, O_I, ψ_j)**, prevede che le due tabelle siano già ordinate ed esegue una scansione parallela.



8.4.4 Group By ed esempi

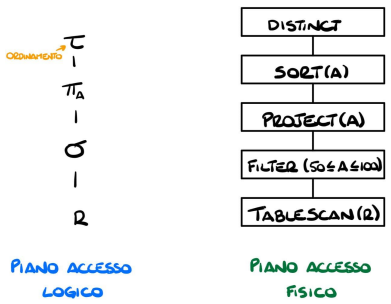
Group By è l'operatore per il raggruppamento e si traduce come la *Distinct* con una coppia di due operazioni distinte: sort e *Group By* consecutivo che funziona in questo modo:

- chiede il next;
- il *Group By* consecutivo comincia a fare una sequenza di next al suo operando che si interrompe solo nel momento in cui trova un break sull'attributo di raggruppamento.



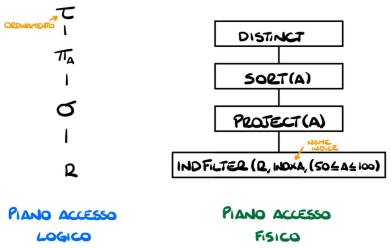
Esempi. Generazione di piani di accesso da query:

```sql
SELECT DISTINCT A
FROM R
WHERE A BETWEEN 50 AND 100
ORDER BY A
```



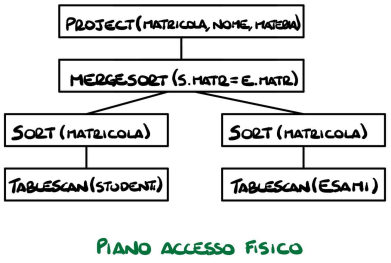
- Supponendo esiste un indice su A

```sql
SELECT DISTINCT A
FROM R
WHERE A BETWEEN 50 AND 100
ORDER BY A
```

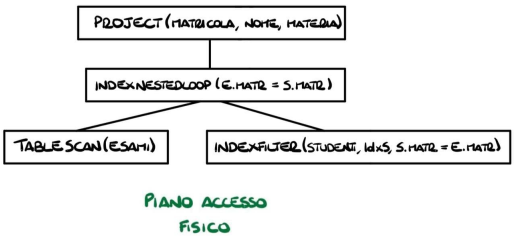


- Senza indici

```sql
SELECT matricola, nome, materia
FROM Studenti S, Esami E
WHERE S.matricola = E.matricola
```

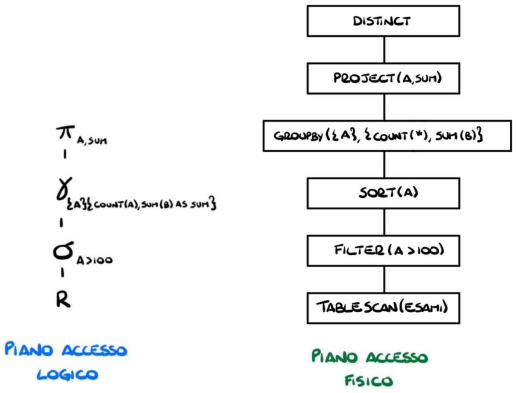


Stessa query ma con indice su S.matricola



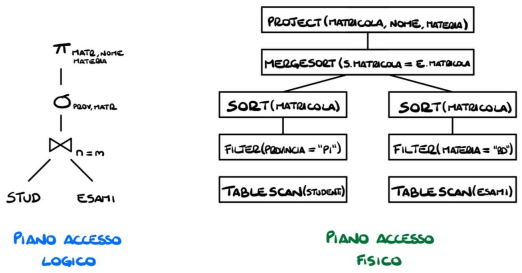
-

```sql
SELECT DISTINCT A, SUM(B)
FROM R
WHERE A > 100
GROUP BY A
HAVING COUNT (*) > 1
```

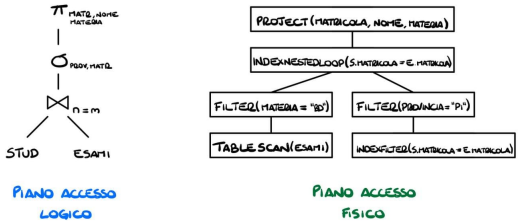


- Senza indici

```sql
SELECT matricola, nome, materia
FROM Studenti S, Esami E
WHERE S.matricola = E.matricola AND provincia = 'PI' AND materia = 'BD'
```



Stessa query ma con indice su S.matricola



8.5 Ottimizzazione delle interrogazioni

L'ottimizzazione delle interrogazioni è fondamentale nei DBMS, ed è necessario conoscerne il funzionamento per una buona progettazione fisica, ovvero per capire quando un indice potrebbe essere utile per eseguire rapidamente una interrogazione. L'obiettivo dell'ottimizzatore è scegliere il piano con costo minimo, fra i possibili piani alternativi, usando le statistiche presenti nel catalogo. L'ottimizzazione funziona in questo modo:

- **Analisi e semplificazione**, il sistema analizza il comando SQL per verificare se è corretto e per apportare alcune semplificazioni, e genera un albero di operatori logici;
- **Trasformazioni**, a volte opera delle trasformazioni sul piano logico;
- **Ottimizzazione fisica**, il compilatore sceglie tra gli innumerevoli diversi piani di accesso fisici quello migliore;
- **Esecuzione piano di accesso**.

8.6 Gestione delle transazioni

Il DBMS compie due lavori importanti: ottimizzare le query e gestire le transazioni. Una transazione dal punto di vista del DBMS è un insieme di operazioni di lettura e scrittura. Una operazione di lettura, comporta la lettura di una pagina nel buffer. Una operazione di scrittura comporta l'eventuale lettura nel buffer di una pagina e la sua modifica nel buffer, ma non necessariamente la sua scrittura in memoria permanente, per questa ragione in caso di malfunzionamento si potrebbe perdere l'effetto dell'operazione. I tipi di malfunzionamento possono essere:

- **Fallimenti di transazioni**, la transazione non funziona correttamente e in questo caso interviene il principio dell'atomicità, cioè deve essere come se questa transazione non fosse mai partita;
- **Fallimenti di sistema**, tipicamente sono errori del sistema che comportano la perdita di dati in memoria temporanea ma non di dati in memoria persistente;
- **Disastri**, sono quei fallimenti nei quali si perdono i dati in memoria temporanea ma si potrebbero perdere i dati anche in memoria persistente. Il difficile è garantire la persistenza.

L'algoritmo standard per proteggere i dati da malfunzionamenti funziona in questo modo:

- a intervalli regolari si copia la base di dati su un altro disco sicuro;
- durante tutto l'uso della base di dati, tutte le operazioni che vengono fatte dalle varie transazioni vengono salvate in maniera sicura su un **giornale** chiamato anche il **log**. Il giornale viene tenuto in doppia copia, cioè tutte le operazioni vengono scritte in due dischi separati. Quando inizia una transazione, si scrive sul giornale:
  - (T, begin) cioè *la transazione T ha incominciato*;
  - Per ogni operazione di modifica si scrive:
    * la transazione responsabile;
    * il tipo di operazione eseguita;
    * la nuova e vecchia versione del dato modificato: (T, write, address, vecchio valore, nuovo valore).
  - (T, commit) o (T, abort), quando una transazione termina. Quando si scrive ciò si garantisce la persistenza.

Per garantire la ripartenza del DBMS quando va giù, oltre a creare una copia della base di dati una volta al giorno, si carica tutto il contenuto del buffer sulla base di dati ogni pochi minuti. In questo modo, in seguito ad un fallimento di sistema, non serve ripartire dalla copia ma direttamente dall'ultimo **checkpoint** della base di dati. Un modo semplice per fare il checkpoint è sospendere l'attivazione di nuove transazioni e completare le precedenti, in seguito si riportano su disco tutte le pagine modificate, si scrive sul giornale il luogo del checkpoint e si riprendono le operazioni. Questo modo però è troppo lento per il fatto che ogni 5 minuti si devono sospendere tutte le attività per 20/30 secondi. Un'altra maniera più complicata ma più interessante, chiamata **disfare - rifare**, è il seguente:

- si scrive sul giornale *(begin checkpoint,* $\{T1, ..., T5\}$*)*, cioè si riporta il luogo di inizio e l'elenco delle transazioni attive;
- in parallelo alle normali operazioni delle transazioni, il gestore del buffer riporta sul disco tutte le pagine modificate;
- si scrive sul giornale il luogo di fine checkpoint;
- il luogo di fine checkpoint certifica tutte le scritture avvenute prima del *begin checkpoint* ora sono sul disco. Le scritture avvenute tra l'inizio e la fine del checkpoint forse sono sul disco e forse no.

Il protocollo di ripresa dai malfunzionamenti, di tipo disfare - rifare, per i vari fallimenti sono:

- *Fallimento di transazioni*, si scrive sul giornale (T, abort) cioè transazione fallita, si cerca sul giornale tutte le operazioni di quella determinata transazione e si applica la procedura disfare;
- *Fallimento di sistema*, si legge il giornale a partire dall'ultimo checkpoint, si scoprono quali sono le operazioni terminate che devono essere rifatte, mentre per le operazioni non terminate devono essere disfatte;
- *Disastri*, si riporta in linea la copia più recente della base di dati, e la si aggiorna rifacendo tutte le modifiche di tutte le transazioni terminate che si trovano sul giornale.

Questo protocollo è il più comune ma esistono degli approcci alternativi:

- **Disfare - NonRifare**, si fa il commit dopo aver portato i dati dal buffer al disco, in questo modo in caso di fallimento di sistema non si ha bisogno di rifare perchè i dati sono già sul disco. Questo protocollo non è molto usato perchè costringe a svuotare il buffer tutte le volte che si fa il commit;
- **NonDisfare - rifare**, per non dover mai disfare si aspetta a scrivere posizionando un *pin* speciale alle pagine, che proibisce al gestore del buffer di portare quella pagina sul disco. Solo dopo il commit si fa un *unpin*. Questo protocollo è più comune rispetto alla precedente però ha un grosso difetto che se una transazione ha bisogno di modificare 100 pagine ma il buffer contiene solo 90 pagine, la transazione dopo aver pinnato 90 pagine va in crash.

Il protocollo del disfare ha il vantaggio che le modifiche possono essere portate nella base di dati stabile prima che la transazione termini, però per poter disfare vale la regola del **write ahead log**: la vecchia versione della pagina deve essere portata prima sul giornale in modo permanente

Esercizio. Si consideri un sistema gestito con protocollo disfare-rifare, si supponga che avvenga un fallimento di sistema, e si assuma che al momento della ripartenza questo sia il contenuto del log, dove un record (W,T1,A,1,10) sta a indicare una scrittura (Write) della transazione T1 sulla variabile A con vecchio valore 1 e con nuovo valore 10:

1. (begin,T1) (W,T1,A,0,20)
2. (begin,T2) (W,T2,B,0,30)
3. (begin-ckp,T1,T2) (W,T1,C,0,30) (end-ckp)
4. (begin,T3) (W,T2,B,30,50)
5. (commit,T1) (W,T3,A,20,50)
6. (begin,T4) (W,T4,D,0,50) (commit,T3)

Richieste:

1. Al momento in cui il sistema aveva iniziato a prendere il log, qual era il valore di A, B, C, D?
2. Qual era il valore di A, B, C, D nel buffer al momento del checkpoint
3. Al termine del checkpoint qual è il valore di A, B, C, D sul disco?
4. Quali sono le transazioni che erano riuscite a dare il commit prima del fallimento di sistema e quali no?
5. Al momento del fallimento di sistema qual è il valore di A, B, C, D nel buffer?
6. Cosa possiamo dire riguardo al valore di A, B, C, D sul disco al momento del fallimento di sistema?
7. Quali sono le transazioni che al momento della ripartenza dovranno essere rifatte e quali sono quelle che dovranno essere disfatte?
8. Si elenchino tutte le operazioni che devono essere rifatte, nell'ordine in cui saranno rifatte
9. Si elenchino tutte le operazioni che devono essere disfatte, nell'ordine in cui saranno disfatte
10. Qual è lo stato delle variabili al termine della ripartenza nel buffer?
11. Qual è lo stato delle variabili al termine della ripartenza nel disco?

Risposte:

1. A = 0, B = 0, C = 0, D = 0
2. A = 20, B = 30, C = (0, 30), D = *no nel buffer*
   dove con (0,30) si intende che è passato da 0 a 30 durante il checkpoint
3. A = 20, B = 30, C = 0-30, D = 0
   dove con 0-30 si intende che può essere 0 o 30 a seconda del momento nel quale il buffer viene raggiunto dal thread che scarica il contenuto del checkpoint
4. Ci sono riuscite: T1, T3 perchè si trova il loro commit nel log
   Non ci sono riuscite: T2, T4
5. A = 50, B = 50, C = 30, D = 50
   lo si ricava dall'ultimo record per ciascuno
6. A = 20-50, B =30-50, C = 0-30, D = 0-50
   A forse era 20 ma se (commit,T1) (W,T3,A,20,50) è riportata sul disco allora è 50. Stesso ragionamento per gli altri.
7. Rifatte: T1, T3
   Disfatte: T2, T4
8. Vengono rifatte tutte le operazioni di T1, T2 che sono avvenute dopo il *begin checkpoint* nell'ordine in cui si trovano sul log: (W,T1,C,0,30), (W,T3,A,20,50)
9. Le operazioni da disfare sono quelle fatte da T2 e T4 ma nell'ordine inverso: (W,T4,D,0,50), (W,T2,B,30,50), (W,T2,B,0,30). Viene disfatta anche (W,T2,B,0,30), perchè, anche se è avvenuta prima del checkpoint ha riportato i suoi effetti sul disco
10. Quando si va a disfare: D = 0, B = 0
    Quando si va a rifare: C = 30, A = 50
11. A = 20-50, C = 0-30, D = 0-50, B = 0-30-50
    A è 20-50 perchè prima del fallimento era 20 o 50. Nella ripartenza ha portato nel buffer A = 50, quindi se il buffer è stato scaricato durante la ripartenza allora A = 50, se il buffer durante la ripartenza non è stato scaricato allora A rimane nello stato che era prima del fallimento quindi 20 o 50. Stesso ragionamento per C e D. B può essere 0 perchè durante la ripartenza l'effetto del disfare è stato portato sul disco, oppure è rimasto come prima perchè il buffer si è tenuto l'effetto del disfare in buffer.

8.7 Gestione della concorrenza

L'esecuzione concorrente di transazioni è essenziale per un buon funzionamento del DBMS. Il DBMS deve però garantire che l'esecuzione concorrente di transazioni avvenga senza interferenze in caso di accesso agli stessi dati. Una esecuzione è **seriabile** se per ogni coppia di transazioni T_1, T_3, tutte le operazioni di T_1 vengono prima di T_3, o viceversa. L'esecuzione seriale è troppo lenta, per questo motivo si vuole una esecuzione che sia **serializzabile**, ovvero che le transazioni hanno il massimo di parallelismo ma l'effetto sulla base di dati è lo stesso di quello ottenibile eseguendo le transazioni serialmente. Per raggiungere questo obiettivo si eseguono tutte le transazioni attraverso un **gestore della concorrenza** che gestisce delle entità astratte che si chiamano **lock**. I lock sono dei permessi. Nel meccanismo che viene utilizzato per gestire la concorrenza vi sono due attori: le transazioni e il serializzatore. Il protocollo del lock a due fasi si basa su tre principi:

- ogni transazione, prima di effettuare una operazione chiede il permesso al serializzatore;
- il serializzatore non fornisce mai a due transazioni diverse due lock in conflitto. Se il serializzatore ha dato il permesso a T_1 di leggere la variabile A, e la transazione T_2 vuole scrivere su A, quest'ultima transazione non riceverà il permesso fintanto che T_1 non restituirà il permesso. Il serializzatore quando T_2 chiede il permesso di scrivere su A può non rispondere, mettere T_2 in stato di wait se ne ha il potere, o abortirlo;
- la transazione anzichè rilasciare il lock immediatamente al termine della operazione, lo tiene e lo rilascia soltanto al termine della transazione.

Il protocollo è a due fasi perchè vi è una fase di acquisizione, cioè quando la transazione viene eseguita, e una fase di rilascio che avviene tutto insieme durante il commit o l'abort. Implementazione standard: tutte le volte che una transazione chiede un permesso che non può avere, mette quella transazione in uno stato di attesa. La situazione di stallo di quando due transazioni sono in attesa l'una di un lock che verrà rilasciato dall'altro si chiama **attesa circolare**. Il problema dello stallo si può risolvere in due modi:

- **Deadlock prevention**, quando una transazione giovane chiede un lock che è posseduto da una transazione più vecchia, anzichè essere messa in attesa viene abortita. Non viene molto usato perchè prevede una quantità eccessiva di aborti;
- **Deadlock detection and recovery**, è una tecnica a posteriori cioè quando rivelano una situazione di stallo, la sbloccano facendo abortire una o più transazioni in attesa. Due esempi sono:
  - **Timeout**, se una transazione aspetta un lock per più di 60 secondi, la si considera come una *detection* di deadlock, e quindi la transazione viene abortita;
  - **Grafo delle attese**, il serializzatore gestisce un grafo: se T_1 viene messo in attesa di un lock posseduto da T_2 si aggiunge un arco da T_1 a T_2. Il serializzatore ogni tanto controlla che non ci sono cicli nel grafo e nel caso ci fossero sceglie una transazione e la fa abortire.