

# Base di dati

Marco Leto - MAT 209645

Anno Accademico 2021/2022

# Contents

<b>1</b>	<b>Introduzione al corso</b>	<b>4</b>
<b>2</b>	<b>Cosa è una base di dati</b>	<b>6</b>
2.1	Creazione di una base di dati . . . . .	6
<b>3</b>	<b>Modello Entità/Relazione</b>	<b>10</b>
3.1	Generalizzazione . . . . .	13
<b>4</b>	<b>Modello Logico</b>	<b>17</b>
4.1	Definizione di Chiave Candidata . . . . .	22
4.2	Definizione di Chiave Esterna . . . . .	23
4.3	Generalizzazione nel modello logico . . . . .	26
4.4	ISA nel modello logico . . . . .	29
<b>5</b>	<b>Interrogazione sulle relazioni</b>	<b>30</b>
5.1	Selezione . . . . .	30
5.2	Proiezione . . . . .	31
5.3	Prodotto Cartesiano . . . . .	33
5.4	Join . . . . .	34
5.5	Differenze tra Join e Sottrazione . . . . .	36
5.6	Interrogazione particolari . . . . .	37
<b>6</b>	<b>SQL</b>	<b>42</b>
6.1	Join in MySQL . . . . .	45
6.2	Operazioni Algebra relazionale . . . . .	45
6.3	Creazione Liste . . . . .	46
6.4	Interrogazioni . . . . .	46
6.4.1	IN . . . . .	47
6.4.2	Linguaggio Dichiarativo . . . . .	49
6.4.3	EXIST . . . . .	49
6.4.4	Aggregazione . . . . .	52
6.4.5	Group By . . . . .	53
<b>7</b>	<b>Dipendenza Funzionale</b>	<b>57</b>

<b>8</b>	<b>Implementazione in memoria delle Basi di Dati</b>	<b>73</b>
8.1	Hashing Lineare a Indirizzamento Aperto . . . . .	78
8.2	B-Tree . . . . .	79
8.2.1	Caratteristiche B-Tree di grado $m$ . . . . .	81
8.2.2	Inserimento nel B-Tree . . . . .	81
8.2.3	Eliminazione di un nodo . . . . .	84
8.2.4	Tuple indicizzabili e Altezza . . . . .	86
8.3	$B^+$ Tree . . . . .	87
<b>9</b>	<b>Modello di Dati Semi-Strutturato</b>	<b>90</b>
9.1	XML . . . . .	91
9.1.1	Elemento Misto . . . . .	95
9.1.2	Albero del documento . . . . .	95
9.2	XPath . . . . .	96
9.3	XQuery . . . . .	99
<b>10</b>	<b>Transazione</b>	<b>106</b>
10.1	Isolamento . . . . .	108
10.2	Schedule Serializable . . . . .	109
10.2.1	Conflict Serializable . . . . .	109
10.2.2	View Serializable . . . . .	111
10.3	Lucchetti . . . . .	113
10.3.1	2 Phase Locking . . . . .	115

# 1 Introduzione al corso

Le basi di dati sono dei sistemi creati per la memorizzazione di dati e che durano per anni. Fondamentale è il concetto di **SERIALIZZAZIONE** che si occupa di trasformare l'oggetto di una classe in una serie di BIT, secondo una codifica e memorizzarli nella memoria secondaria.

Nelle basi di dati è importante tenere in considerazione che si possono creare problemi legati alla "concorrenza" che in Java sono gestiti con semafori e monitor.

Nel corso di sistemi operativi abbiamo inoltre visto che la memoria centrale dei calcolatori utilizza dei meccanismi di mutua esclusione per la gestione dei processi.

Il problema si crea nel momento in cui tali meccanismi lavorano in memoria centrale, mentre nelle basi di dati si lavora sugli hard disk e non sulla RAM.

È importante inoltre che i meccanismi di mutua esclusione non siano pesanti, ciò provocherebbe tempi di attesa per compiere le operazioni.

Uno dei fenomeni che si viene a creare quando più persone accedono allo stesso dato è detto **RACE CONDITION**

$U_1$	$U_2$
read(A)	read(A)
A=A+10	A=A+100
write(A)	write(A)

Supponiamo di avere due utenti che contemporaneamente portano alla modifica della variabile A.

A inizialmente vale 200 e alla fine dovrà valere 310.

Fondamentale sapere che data una risorsa, a tale risorsa, deve accedere, ad un determinato istante di tempo, un solo utente.

Consideriamo di assegnare gli istanti di tempo nel seguente modo:

1	$U_1$
2	
3	$U_2$
4	
5	$U_1$
6	
7	$U_2$
8	

Nell'istante 1-8 hanno lavorato entrambi gli utenti ma non contemporaneamente, un'esempio di possibili operazione è il seguente:

$U_1$	$U_2$
read(A)	
	read(A)
A=A+10	
write(A)	
5	$U_1$
6	
7	$U_2$
8	

Si nota seguendo passo passo quanto scritto che il risultato finale di A sarà 300. Infatti l'Utente 2 legge A=200 e non A=210, non è quindi garantita la mutua esclusione.

Vedremo che per questo tipo di operazioni si utilizzano dei metodi detti **SYNCRONIZED**.

Nelle basi di dati è possibile utilizzare diverse collezioni in base al dato da ricercare. Questo strutture dati si chiamano **INDICI**.

É importante definire la differenza tra dato e informazione che NON sono sinonimi.

Un dato è una rappresentazione di una informazione fatta mediante codifica. Negli elaboratori vengono quindi codificate informazioni, la codifica più semplice è la codifica BINARIA. Una informazione è un concetto PRIMITIVO, che non può essere definito.

Nella TEORIA DELL'INFORMAZIONE si definisce la **MISURA** dell'informazione, ovvero quanto viene alterato lo stato di chi la riceve rispetto al momento prima della ricezione.

## 2 Cosa è una base di dati

Una base di dati è un sistema che serve a gestire collezioni di dati, garantendo:

- DURABILITÀ dei dati
- ACCESSO CONCORRENTE di più utenti contemporaneamente. Tale gestione viene fatta in automatico dalla base di dati.

I sistemi per la gestione delle basi di dati sono i DBMS (DataBase Management System), alcuni esempi sono Oracle, DB2, MySQL(opensource). Access ad esempio, non è una base di dati, questo va a creare un file con una sua estensione e di fatto rappresenta una "maschera" che ci permette di vedere dei dati, il file viene inoltre assegnato interamente per cui può essere modificato da un utente alla volta. Nei DataBase invece l'accesso alle porzioni del file system dedicato ad ospitare i nostri dati, sono gestiti dal DBMS.

### 2.1 Creazione di una base di dati

Lo STUDIO DI FATTIBILITÀ è il primo passo ed è caratterizzato dall'incontro tra committente e progettista, in cui il cliente descrive le necessità che ricerca nella sua base di dati. È una fase in cui anche senza entrare nel dettaglio, è necessario raccogliere informazioni sufficienti per capire se il lavoro può essere fatto o meno.

La fase successiva è l'ANALISI DEI REQUISITI dove si cercano di capire le specifiche, che il committente si aspetta di avere nella sua base di dati. Le specifiche si dividono in:

- funzionali
- non funzionali

Si passa poi alla fase di PROGETTAZIONE CONCETTUALE, dove si "elencano" le informazioni da rappresentare. Questa fase produce un PROGETTO CONCETTUALE rappresentato da un Modello Entità/Relazione.

Segue la PROGETTAZIONE LOGICA/FISICA che serve a definire la struttura logica dei dati più adatta a rappresentare una determinata informazione. Si utilizza il Modello Relazionale che è rappresentato da TABELLE.

## ESEMPIO

Supponiamo di dover compiere la progettazione logica di un supermercato dove bisogna tenere traccia dell'acquisto di prodotti da parte dei clienti. Si possono identificare due configurazioni:

### CASO 1

Cfcliente	NomeCliente	codicePr	descrizPr	qta	prezzo	data
C1	N1	P1	D1	10	20	03-gen
C1	N1	P2	D2	5	10	04-gen
C2	N2	P1	D1	50	100	02-gen

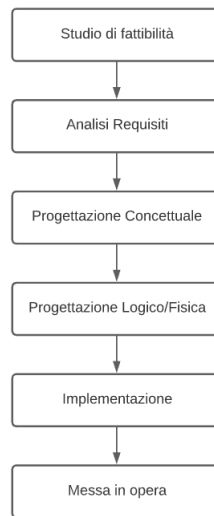
### CASO 2

Cfcliente	descrizPr
C1	N1
C2	N2

codicePr	descriz
P1	D1
P2	D2

Ciente	Prodotto	qta	prezzo	data

Se si volessero riassumere i passi per la costruzione di una base di dati:



Viene chiamata costruzione **WATERALL** o a cascata, ma il termine non è propriamente esatto, infatti spesso andando avanti nella progettazione ci si accorge che sono stati fatti errori o bisogna modificare ciò che è stato fatto nelle fasi precedenti.

Supponiamo di avere dopo l'analisi dei requisiti le seguenti specifiche.

- Inserimento, modifica, cancellazione, ricerca di fornitori
- Caratteristiche dei fornitori(PIVA, nome, ...)
- Inserimento, modifica, cancellazione, ricerca di merce
- Caratteristiche della merce(codice, nome, marca, ...)
- Inserimento, modifica, cancellazione, ricerca di forniture
- Chi fornisce cose e a che prezzo
- Ricerca in base al prodotto e al fornitori

Ogni concetto a se stante si rappresenta in un diagramma con un rettangolo, etichettato con il nome del concetto che può essere arricchito dagli attributi che caratterizzano il concetto, ogni concetto prende il nome di **ENTITY SET**.

Sulla base dell'analisi dei requisiti sarà necessario rappresentare informazioni sui fornitori sulle merci.

La fornitura è una **RELAZIONE** tra i concetti di fornitore e merce, e viene indicata con un rombo. Il rombo collega due Entity Set e i suoi attributi sono dati da un set di coppie. Una relazione



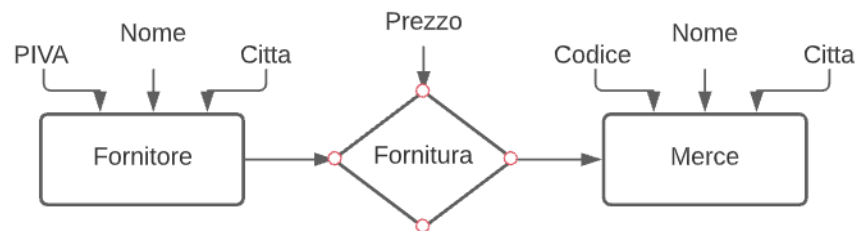
matematica fra due insiemi è un sott'insieme del prodotto cartesiano di due insiemi. Nel caso in analisi dalla coppia di attributi <fornitore, merce> allungata da un attributo che caratterizza solo la fornitura, ovvero il prezzo.

Si capisce il motivo per il quale il modello prende il nome di Entità/Relazione.

Per ogni Entry Set è importante determinare gli attributi che ne garantiscono l'identificazione. Ad esempio nel caso dei Fornitori è la "PIVA", tale attributi prendono il nome di **CHIAVE**. Avere gli attributi "identificatori" é fondamentale, poiché si dice al sistema che non possono essere aggiunte entità il quale attributo identificatore è già presente.

Prese ora le merci, la sintassi corretta per descriverle è la seguente:

"Una merce è caratterizzata da codice, nome e marca ed è identificata dal codice"



Nelle progettazioni di base di dati non capita però spesso di avere una chiave caratterizzata da un solo attributo, per cui la chiave è un **INSIEME** di attributi.

Se volessimo dare una definizione:

"Una chiave è un insieme di attributi che identificano univocamente le entità di una Entity Set, tale insieme deve essere **MINIMALE** rispetto alla proprietà di essere identificabile. Minimale non vuol dire minima, possono infatti convivere più chiavi candidate che hanno cardinalità diversa e tutte le chiavi vanno indicate(nel caso del fornitore una chiave che include tutti gli attributi rappresenta una chiave minimale)."

### 3 Modello Entità/Relazione

Il modello entità relazione è un modello grafico che serve a rendicontare la progettazione concettuale. Non tutto è esprimibile utilizzando il linguaggio grafico, per cui spesso viene integrato con il linguaggio testuale.

Dentro una classe ci può stare un oggetto di un'altra classe, qui invece tutti gli attributi che inseriamo non sono altre istanze di Entity Set, ma sono semplici.

L'allineamento delle informazioni rispetto alla realtà prende il nome di **CONSISTENZA**.

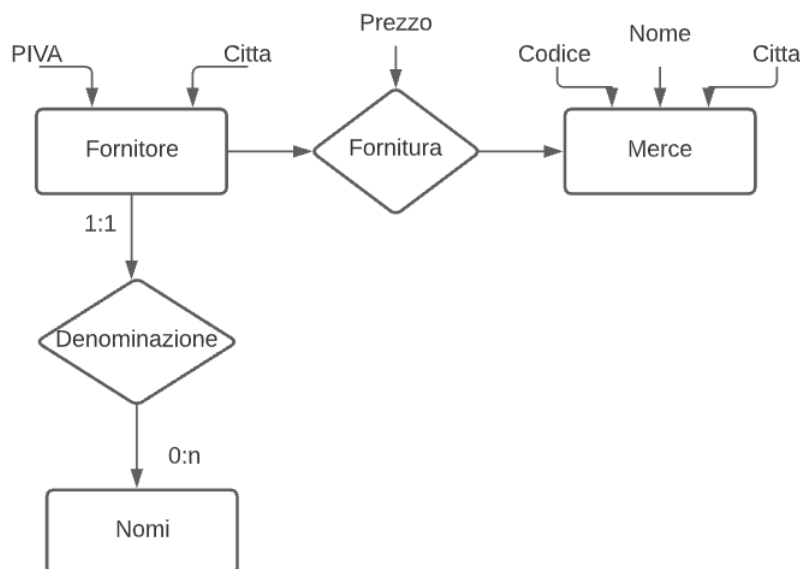
Un dato è integro quando rappresenta correttamente la realtà.

Per ogni entità deve essere indicata almeno una chiave, se ci si rende conto che non è presente, vuol dire che esiste qualche caratteristica dell'Entity set che non è stata considerata.

Per le relazioni invece NON bisogna indicare una chiave, in quanto una istanza di relazione è già identificata dalle istanze delle Entity set che la caratterizzano.

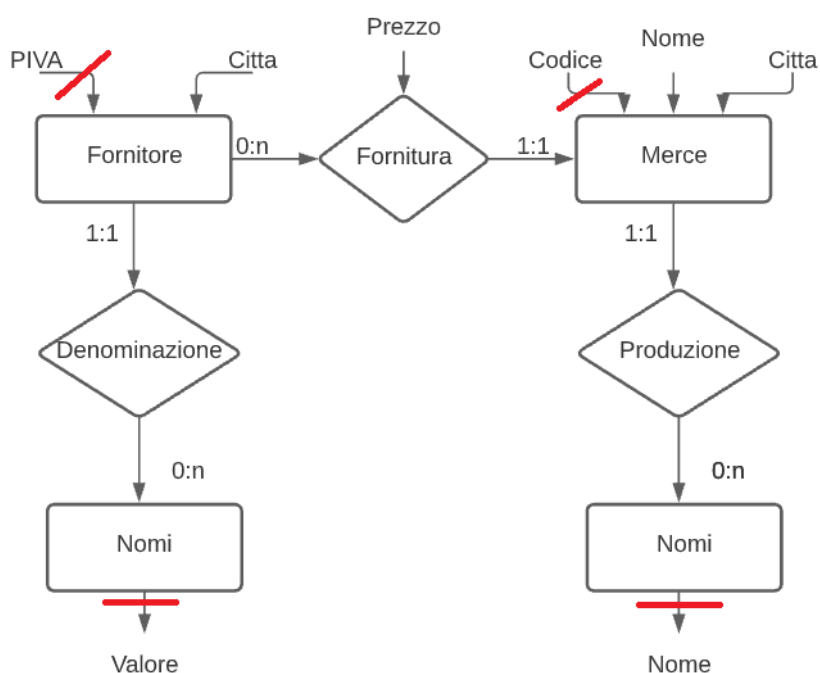
Ogni qual volta si indica una relazione tra due insiemi, si può indicare in maniera specifica in che modo una Entity set contribuisce alla relazione, tale informazione è detta **VINCOLO DI CARDINALITÀ**.

Dato questo concetto possiamo eliminare ad esempio l'attributo di "nome" e creare una relazione chiamata Denominazione tale per cui un fornitore può avere massimo un nome, ma un nome può essere scelto da n fornitori.



La scelta di mettere una entità o un attributo è data dal fatto di consistenza di un dato, se vogliamo che il nome debba appartenere ad una lista di nomi italiani può essere utile metterlo.

Analogamente può avere senso inserire una relazione tra "Merce" e "Marca", eliminando "Marca" dagli attributi. In questo modo può essere inserita solo una "Marca" presente nell'Entity set delle marche. Ciò risulta utile se il committente indica che spesso effettuerà ricerca per "Marca". Questo perché creare una Entity set "Marca" elimina la possibilità di avere errori di inserimento, supponiamo che su 10 volte che viene inserito un prodotto "Barilla", venga battuto una volta "Barillà", nella ricerca per marca "Barilla" non troveremo tale prodotto, inserendo la relazione dovremo preoccuparci di inserire la marca correttamente solo la prima volta.



Il grado di cardinalità 0:n è da discutere, in quanto stiamo indicando che possiamo avere un fornitore che non fornisce alcuna merce. Mettendo invece come vincolo di cardinalità 1:n, inseriamo un vincolo pratico, in quanto non è possibile caricare i dati di fornitori e merci in momenti diversi.

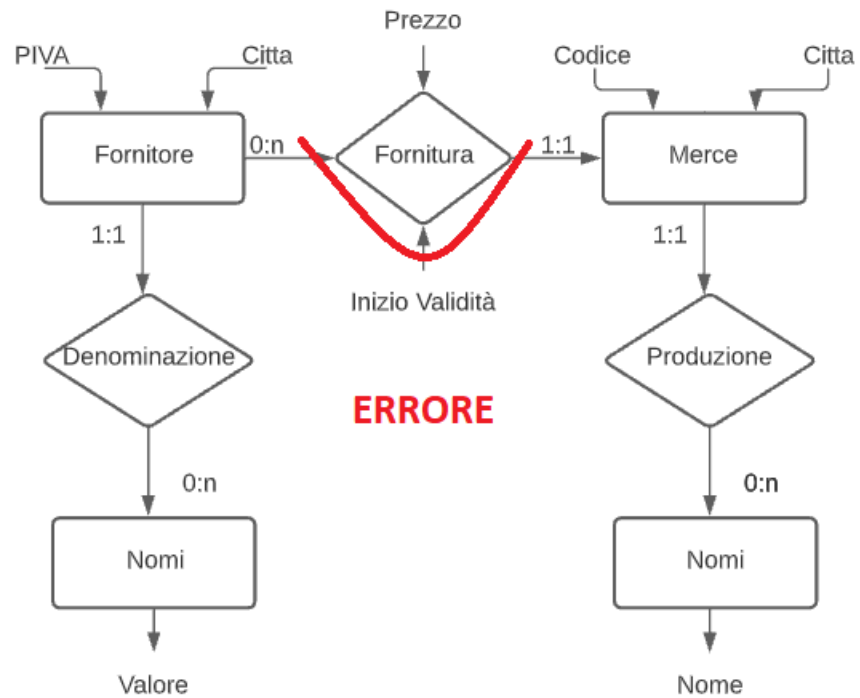
Quando una chiave comprende sia attributi che bracci di relazioni viene detta **CHIAVE MISTA**. Se tutti i membri della chiave sono bracci di relazioni la chiave è detta **ESTERNA**.

È importante definire, che le relazioni che costituiscono le chiavi devono avere un vincolo di cardinalità 1:1, non è però detto che una relazione 1:1 debba afforza appartenere ad una chiave.

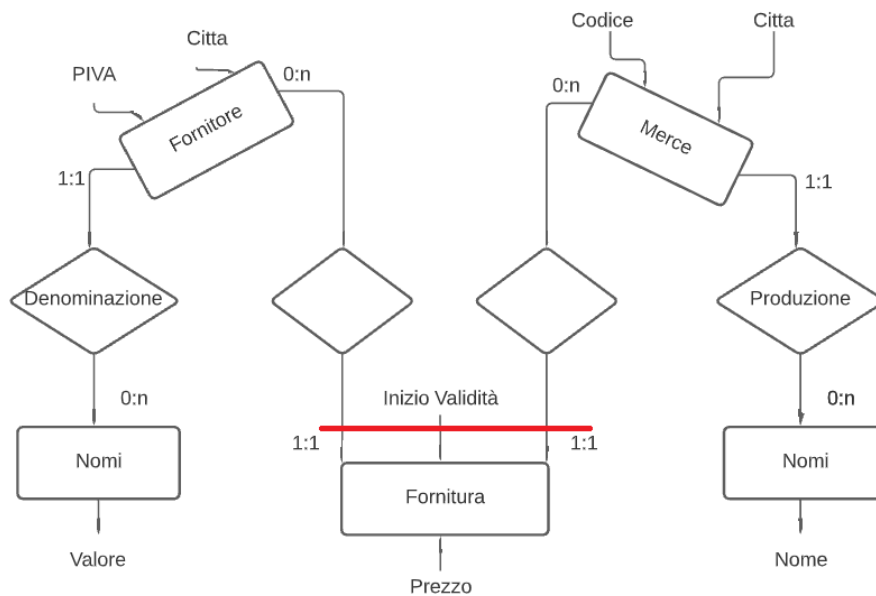
Supponiamo che il committente voglia anche la cronologia delle forniture, e inseriamo nel modello ER un nuovo attributo alla fornitura: "inizio validità"

Una fornitura sarà identificata da una quaterna <Fornitore, merce, inizio validità, prezzo>, non

possiamo però creare una chiave che la identifica in quanto la fornitura è una relazione.



Una soluzione può essere quella di creare una Entity set "Fornitura" e definire la chiave data dalla terna fornitore, merce, inizio validità.



In generale una relazione 0:n 0:n può essere sempre tradotta in una Entity set con due relazioni entrambe con cardinalità 0:n 1:1, come in questo caso. Inoltre una terna di concetti rappresentati da 3 entità la possiamo sempre rappresentare come una nuova entità identificata da una chiave esterna.

### 3.1 Generalizzazione

Supponiamo di trovarci in una situazione in cui dobbiamo rappresentare una base di dati per l'università data da "Professori" e "Studenti", entrambi condividono la caratteristica di essere persone(per cui di tutti prendiamo i dati anagrafici).

Partiamo dalle seguenti specifiche:

- Gestire l'anagrafe di docenti e studenti
- Gestire le docenze(insieme di materie)
- Gestire gli esami(in che data uno studente fa l'esame)

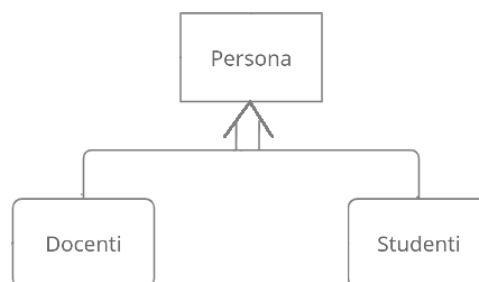
Le proprietà delle due Entity Set vengono messe a fattore comune mediante una generalizzazione la cui entità genitore ("Persona") rappresenta tutte le informazioni anagrafiche di "Studenti" e "Docenti".

Dopodiché le Entry Set vengono gestite in modo classico omettendo naturalmente gli attributi che già vengono ereditati.

Gli "Studenti" sono caratterizzati dal corso di laurea e dalla matricola mentre i "Docenti" dal dipartimento. Le chiavi saranno date dagli attributi delle entità genitore.

Ogni entità eredita tutte le chiavi dall'entità genitore ma a sua volta può offrire altre chiavi candidate(ad es. la matricola)

Date queste informazioni, il diagramma E/R è il seguente:



Una generalizzazione può essere:

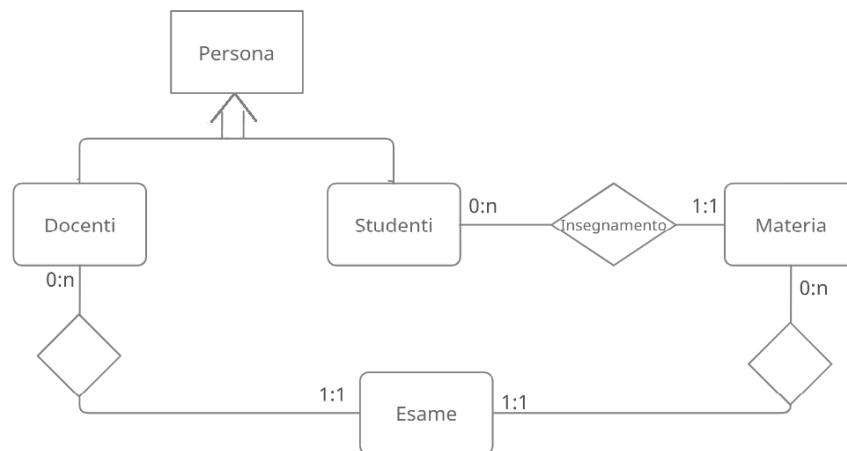
- **TOTALE**
- **PARZIALE**

o ancora:

- **ESCLUSIVA**
- **INCLUSIVA**

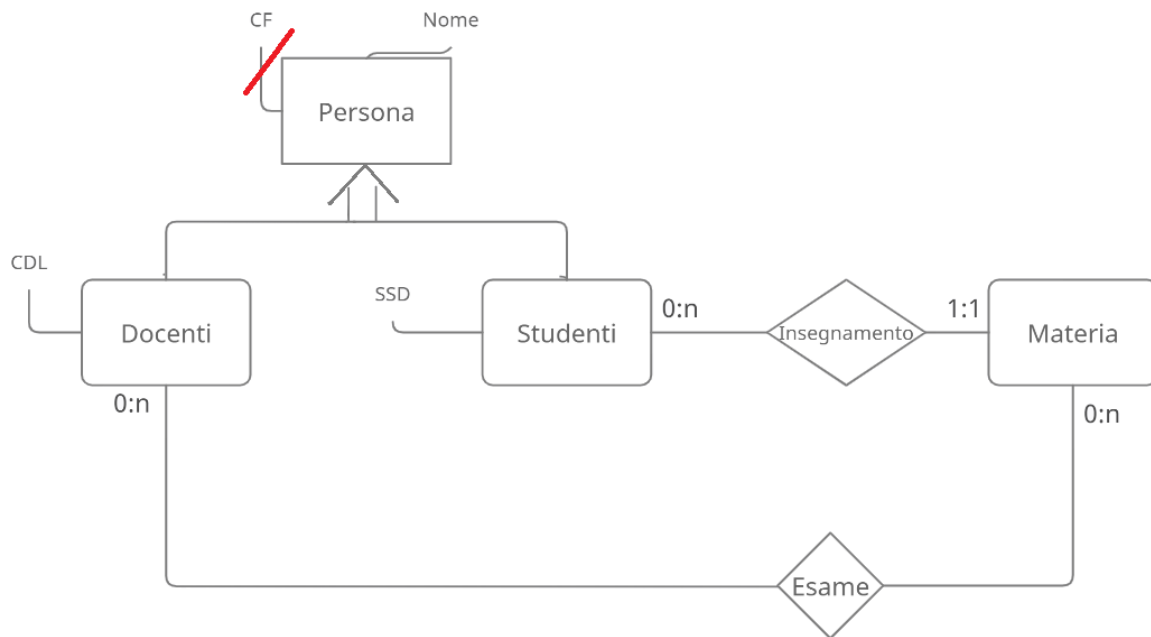
Nel caso specifico discusso, si ha una relazione **TOTALE**, in quanto tutte le istanze delle Entry Set "Studente" e "Docente" prendono parte alla relazione. L'appartenenza ad una delle sotto classi è inoltre una caratteristica **ESCLUSIVA**, in quanto non ci possono essere studenti che sono anche docenti e viceversa. La generalizzazione totale e esclusiva corrisponde ad una partizione, ovvero l'unione fornisce l'insieme totale e l'intersezione l'insieme vuoto. Se si ha questo caso il tipo di relazione generalizzante non va indicato sul diagramma, in tutti gli altri casi sì. Tornando alla base di dati, si sceglie di inserire una Entity Set "Materia", caratterizzata da un codice, da un nome. "Docente" e "Materia" sono legati da una relazione "Insegnamento".

Infine si inserisce la relazione "Esame" data dalla relazione tra "Studente" e "Materia", caratterizzata da data e voto. Entrambi le Entity Set avranno cardinalità 0:n nei confronti della relazione (Uno studente può sostenere n esami così come una materia può sostenere n appelli). È importante indicare che la coppia <studente,materia>, contraddistingue il superamento di un esame e non può esistere una sola occorrenza di questo tipo.



Se volessimo invece registrare tutti i tentativi di prova di un esame, non si può più seguire questo schema in quanto la coppia di entità non contraddistingue più un esame.

Si può quindi pensare immediatamente di caratterizzare la chiave con un attributo su esame, ma ciò è errato per quanto detto precedentemente, si può però creare una nuova Entity Set "Esame" legata a "Studente" e "Materia" con delle relazioni.

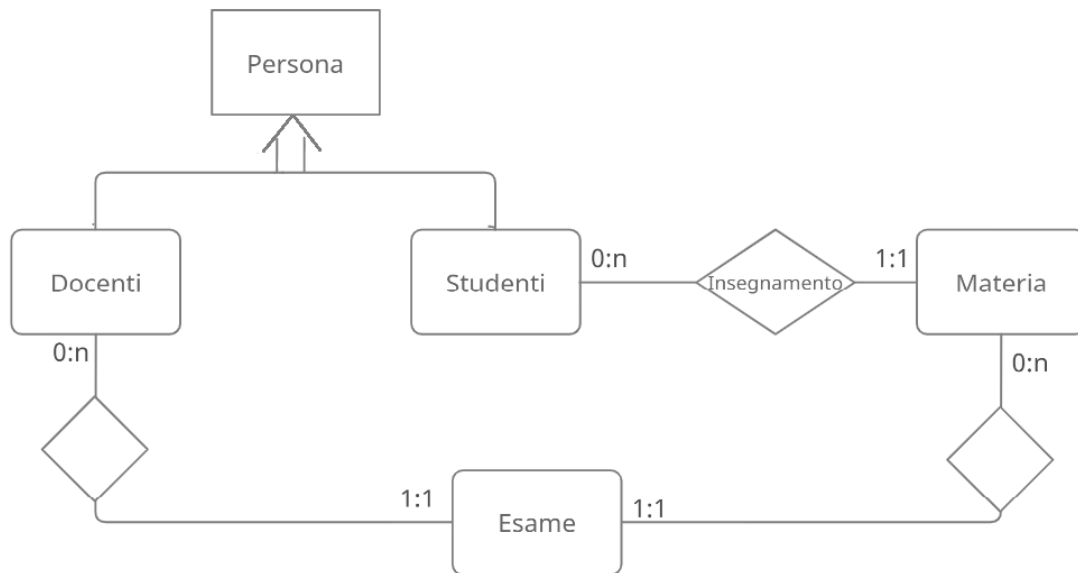


Questa configurazione rende però possibile sostenere più volte esami e prendere voti diversi, ma un esame può essere superato una sola volta. Questo vincolo non può essere espresso logicamente ma andrà inserito testualmente nella base di dati.

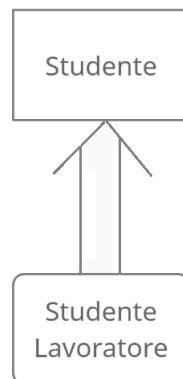
Tornando alla generalizzazione, questa può essere realizzata tra più entità:

Inoltre si può individuare un **caso particolare**, in cui si ha una generalizzazione con una sola sotto-entità. Ad esempio si presuppone di avere un insieme di studenti caratterizzati da una matricola ed un nome, e un'altra entità dati dagli studenti lavoratori, che sostanzialmente sono studenti che integrano con alcuni attributi.

Il modello E/R sarebbe il seguente:



Tale relazione è anche detta "Is a". Le due Entity Set potrebbero poi avere diverse relazioni, ricordando sempre che le relazioni di "Studente" sono anche relazioni di "Studente Lavoratore" ma non viceversa.



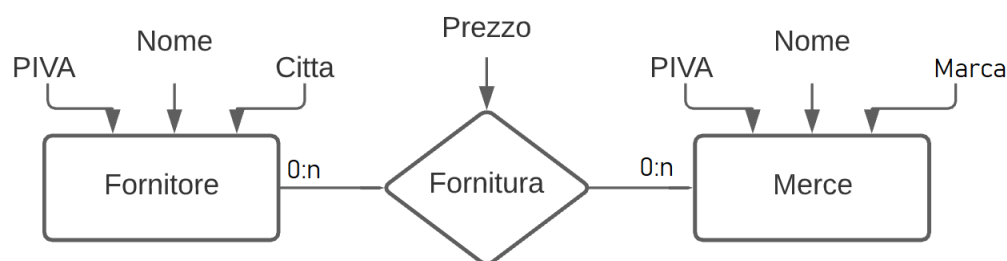


## 4 Modello Logico

In questo corso vengono studiati i Database Relazionali, Relazionali indicano che viene definito un modello logico, ovvero la **tabella**. Tutte le informazioni vengono identificate da un insieme di righe(record), che nell'ambito dei database prende il nome di **TUPLA**.

Quando viene fatta la progettazione logica non è necessario scegliere la struttura dati da utilizzare. La tabella a sua volta ha sicuramente delle strutture di memorizzazione ma a chi programma la base di dati, non interessa.

Si riprende il modello visto precedentemente:



Un esempio di tabella potrebbe essere il seguente:

PIVA	nomeF	citta	codice	nomeM	marca

Generalmente non si realizza la tabella, poiche risulta scomodo e poco pratico. Le tabelle vengono così dichiarate:

NomeTabella(PIVA,nomeF,citta,codice,nomeM,marca)

Non si è vincolati a fare un'unica tabella in quanto un modello logico è una relazione tra più tabelle. Simulando una pseudo compilazione della tabella otteniamo la seguente **ISTANZA DELLA TABELLA** con 4 tuple:

PIVA	nomeF	citta	codice	nomeM	marca
F1	N1	C1	M1	NM1	alfa
F1	N1	C1	M2	NM2	beta
F2	N2	C2	M1	NM1	alfa

Questa scelta di tabella non risulta però opportuna in quanto si presentano:

- Problemi di Ridondanza Devo ripetere gli stessi dati per ogni merce fornita dallo stesso fornitore. Il fenomeno potrebbe provocare la poca efficienza nell'aggiornamento dei dati di un fornitore, che magari cambia città ed è necessario cambiare il dato in 100 tuple. E ancora, supponiamo di aggiungere una nuova tupla:

F2	N2	C2	M2	NM3	beta
----	----	----	----	-----	------

Avremo due volte la merce con codice M2, fornita da due fornitori diversi, che presenta nome diverso, e questo è chiaramente un errore, in quanto il codice di una merce la identifica univocamente. Per cui questa configurazione costringe ad utilizzare dei meccanismi di controllo che evitino inconsistenze e che rallentano la base di dati.

- Siccome stiamo rappresentando le forniture, con questa rappresentazione ci possono essere merci che non stanno in fornitura e non possono essere rappresentate. In realtà possiamo utilizzare tuple pseudo-compilate con valori null:

F3	N3	C2	null	null	null
null	null	null	M3	NM3	theta

Quando un fornitore è associato ad delle colonne null , vuol dire che non fornisce alcuna merce e ancora quando una merce è associata a delle colonne null vuol dire che non è associata ad alcun fornitore.

Questo è un modo necessario per poter rappresentare la cardinalità 0:n espressa

- Il modello relazionale ci dice che non possono esistere due fornitori con stessa PIVA e due merci con stesso codice. Inoltre ogni tupla della tabella è identificata dalla coppia {fornitore, merce}

che deve essere univoca, ma nessuno vieta l'inserimento di una tupla del genere (consideriamo anche l'attributo prezzo):

F1	N1	C1	M1	NM1	alfa	10\$
.						
.						
.						
.						
F1	N1	C1	M1	NM1	alfa	20\$

In questo senso il modello relazionale ci viene in aiuto, in quando è presente un **Vincolo di chiave**, che nel modello entità relazioni va indicata solo nella relazione, mentre nel modello relazionale, per ogni tabella che viene indicata deve essere indicata la rispettiva chiave.

Una chiave è un vincolo espresso da un insieme di attributi.

Questa tabella di fatto traduce il concetto di fornitura, per cui non possono esistere più righe che coincidono sia in PIVA che in codice. Indicare questo vuol dire che non sarà ammesso popolare la tabella con due tuple con gli attributi sopra citati uguali.

Definire uno **SCHEMA DI UNA TABELLA**, vuol dire:

- dare un nome alla tabella
- indicare l'elenco delle colonne che la comporranno
- definire un meccanismo di autenticazione delle righe

Il modello più opportuno, è un modello in cui sono presenti le sorgenti tabelle:

Fornitori(PIVA,nome,città)

Merce(codice,nome,marca)

Fornitura(Fornitore,merce,prezzo)

F1	N1	C1	M1	NM1	alfa	10\$
.						
.						
.						
.						
F1	N1	C1	M1	NM1	alfa	20\$

É stato risolto il problema della ridondanza.

PIVA	nome	citta
F1	N1	C1
F1	N2	C2

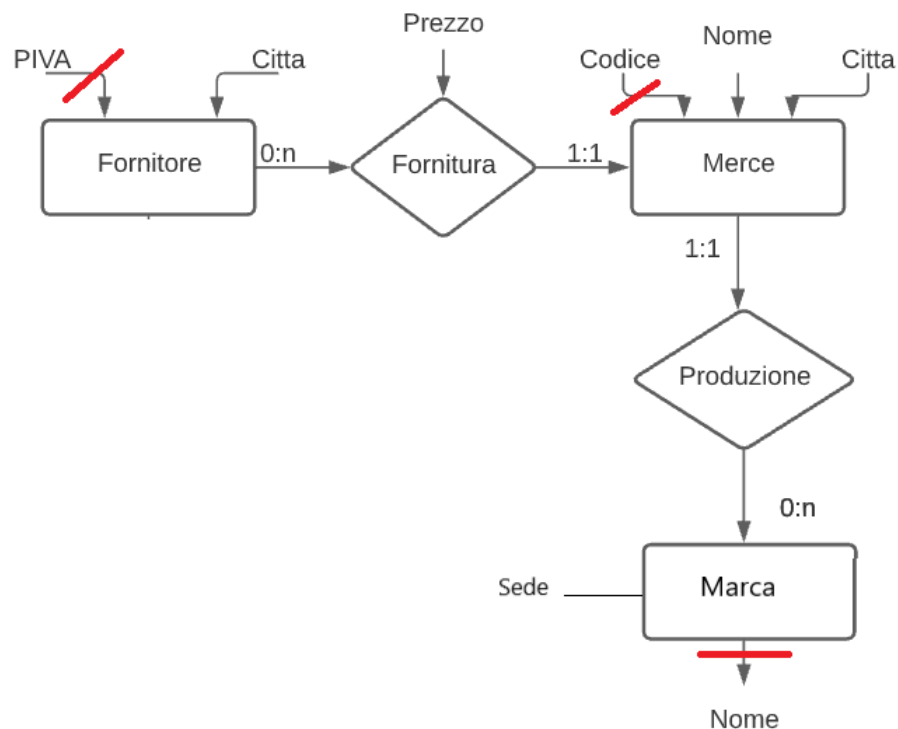
codice	nome	marca
M1	NM1	alpha
M2	Nm2	beta

Fornitore	Merce	prezzo
F1	M1	...
F1	M2	...

In realtà la vera dichiarazione delle tabella richiede che sia inserito anche il tipo di dato(ad es. Varchar che rappresenta il tipo String):

Fornitore(PIVA:Varchar,nome:Varchar(20),città:char(15))

É possibile arricchire il modello E/R rappresentando il concetto che una "Merce" è caratterizzato da una "Marca" da una relazione.



Seguendo la politica per cui viene associata una tabella ad ogni entità avremo rispetto al caso precedente le tabelle:

Fornitori(PIVA, nome, città)

Merce(codice, nome, marca)

Marca(nome, sede)

Fornitura(Fornitore, merce, prezzo)

Produzione(merce, marca)

A primo impatto le chiavi di questa tabella dovrebbero essere merce e marca. Ma non è così in quanto si ha una relazione 1:1 per cui una merce può essere prodotta da una sola marca. Il che significa che nella tabella "Produzione" basta il codice della merce per identificare una coppia <merce, marca>.

È palese che non valga il contrario poiché una marca può produrre 0:n merci.

Mettere la "Merce" esterna in realtà è scorretto, proprio a causa del vincolo 1:1. Poiché viene ammesso che siano merci che stanno in "Merce" e non in "Produzione" (nel caso in cui avessimo avuto cardinalità 0:1, non avremmo avuto questo problema).

Per generalizzare il concetto:

- Se il vincolo è 1:1 non materializzo alcuna tabella che traduce la relazione. Semplicemente inserisco l'attributo.
- Se il vincolo è 0:1 e dall'altra parte 0:n, è possibile ragionare in due modi:
  - Nella tabella:

Merce(codice, nome, marca)

viene inserito marca=null, nel caso in cui la merce non ha marca

- Si crea l'entità esterna "Marca" e la conseguente tabella "Produzione"

In generale se abbiamo poche merci con marche si consiglia di attuare la seconda soluzione, al contrario la prima.

## 4.1 Definizione di Chiave Candidata

Le chiavi di un'entità nel modello E/R prendono il nome di **CHIAVI CANDIDATE**.

**Definizione 1.** *Dato uno schema relazionale*

$$R(A_1, A_2, \dots, A_n)$$

*definiamo chiave (candidata) un sottoinsieme  $K$  non vuoto minimale di  $A_1, \dots, A_n$*

$$K = \{K_1, \dots, K_n\} \subseteq \{A_1, \dots, A_n\}$$

*tale che*

1.  $\forall r$  istanza legale di  $R$ ,  $\forall t_1, t_2 \in r$   
 $t_1 \neq t_2 \implies t_1[K_1] \neq t_2[K_1] \vee \dots \vee t_1[K_m] \neq t_2[K_m]$ .
2.  $\nexists Y \subset K$  t.c. vale 1.

Una chiave è un sottoinsieme non vuoto minimale di attributi che vincola le istanze di uno schema a non contenere tuple distinte che coincidono nel valore degli attributi della chiave.

Nel modello Logico/Fisico, ad ogni tabella è associata una chiave, questa prende il nome di **CHIAVE PRIMARIA**. Se ci sono più candidate a chiave, se ne sceglie una e le altre prendono il nome di **VINCOLI DI UNICITA** (Unique).

Solitamente la chiave primaria è scelta sulla base del fatto che la chiave scelta è il meccanismo più tipico rispetto al quale viene fatta la ricerca nella base di dati.

## 4.2 Definizione di Chiave Esterna

**Definizione 2.** *Dati due schemi relazionali, appartenenti allo stesso schema di DB,*

$$R_1(A_1, A_2, \dots, A_n)$$

$$R_2(B_1, B_2, \dots, B_n)$$

*definiamo vincolo di chiave esterna un'espressione del tipo*

$$R_1[A'_1, \dots, A'_k] \sqsubseteq_{FK} R_2[B'_1, \dots, B'_k]$$

*dove  $A'_1, \dots, A'_k$  è un sottoinsieme di attributi di  $R_1$  e  $K = \{B'_1, \dots, B'_k\}$  è una chiave di  $R_2$ .*

Nel modello logico, la **CHIAVE ESTERNA**, indica che in una tabella un determinato campo non può assumere valori a caso ma deve assumere i valori in corrispondenza della chiave primaria dell'istanza che richiama.

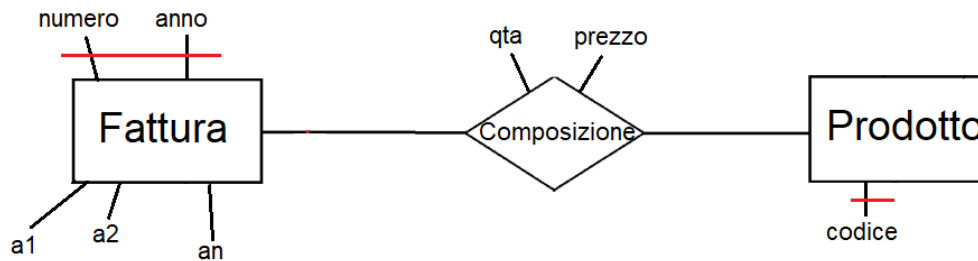
Si chiama chiave esterna, perché uno o più attributi della tabella corrente, fanno riferimento ai valori assunti da una chiave di un'altra relazione. Chiave esterna NON vuol dire che quell'attributo identifichi una chiave per la tabella in analisi.

Un attributo membro di una chiave primaria non può essere null. Per cui una chiave esterna non può essere identificata da una relazione con cardinalità 0:1, in quanto si ammette la possibilità che l'attributo abbia valore "null".

Non è possibile esprimere una relazione di chiave esterna su solo un attributo di una chiave mista.

## ESEMPIO

Si supponga di avere il seguente modello E/R:



Dal quale derivano le tabelle:

Fattura(numero, anno, a1, a2)

Prodotto(codice,...)

Composizione(fattura, prodotto, ...)

Quando si scrive però il vincolo di chiave esterna, fattura prende valori da due colonne ma è rappresentata da una sola colonna, si tratta infatti di un ERRORE.

È necessaria la seguente relazione:

Composizione(fatturaNum, fatturaAnno, prodotto, ...)

Il vincolo di chiave esterna sarà che:

$\text{Composizione}[\text{fatturaNum}, \text{fatturaAnno}] \sqsubseteq_{FK} \text{Fattura}[\text{numero}, \text{anno}]$

Un altro errore può essere il seguente:

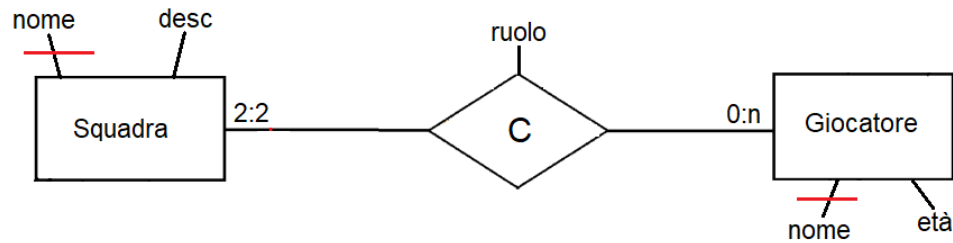
$\text{Composizione}[\text{fatturaNum}] \sqsubseteq_{FK} \text{Fattura}[\text{numero}]$

$\text{Composizione}[\text{fatturaAnno}] \sqsubseteq_{FK} \text{Fattura}[\text{anno}]$

Potrebbe portare al riferimento di un numero di fattura che non corrisponde all'anno in quanto non identifichiamo una coppia ma due valori singolarmente.



Si supponga di gestire delle squadre di Padel:



Il modello logico potrebbe essere il seguente:

Squadra(nome, desc, gioc1, ruolo1, gioc2, ruolo2)

Giocatore(nome, età)

Squadra[gioc1]  $\sqsubseteq_{FK}$  Giocatore[nome]

Squadra[gioc2]  $\sqsubseteq_{FK}$  Giocatore[nome]

Si ammette però il caso in cui i due giocatori di una squadra siano uguali.

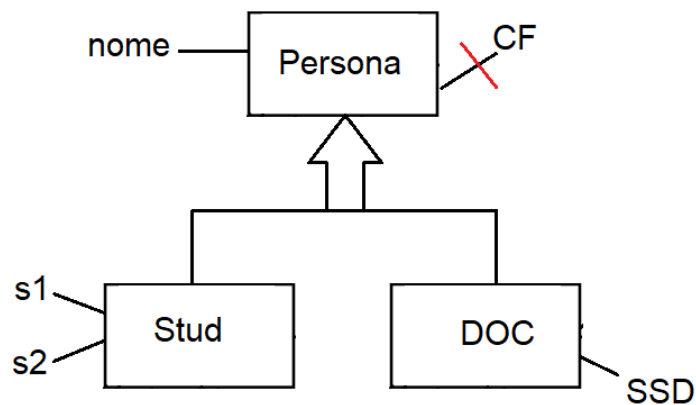
Quando si ha una cardinalità fissata, del tipo (2:2, 3:3, 4:4), si sceglie di inserire più campi nella tabella dell'entità.

Se i numeri iniziano a diventare importanti, dell'ordine di 100:100, si può rilassare la relazione sostituendo la cardinalità con 0:n, anche perché a livello di implementazione, diventa difficilmente gestibile, avere una tabella con 100 colonne.

Quando il vincolo è di tipo 0:2, si ammette che i campi "giocat1" e "giocat2" nella tabella squadra possano avere valore null. Vanno discussi anche gli attributi "ruolo1" e "ruolo2", se si ammette che possano avere valore null, può accadere che lo assumano anche nel momento in cui si ha il "giocat1" e "giocat2". Per sviare a questo problema, nel momento in cui gli attributi di giocatore sono null, il valore contenuto nei ruoli, non viene considerato, qualsiasi esso sia.

### 4.3 Generalizzazione nel modello logico

Si supponga ora di avere una generalizzazione:

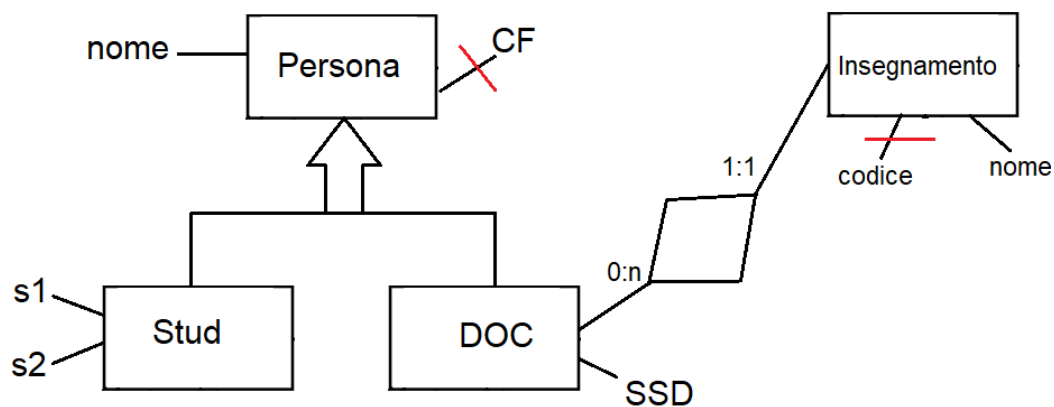


Si possono scrivere le relazioni:

$\text{Persona}(\underline{\text{CF}}, \text{nome}, \text{stud/doc}, \text{SSD}, \text{S1}, \text{S2})$

Ogni persona viene tradotta in una tupla, che contiene un booleano, se è vero si ha uno "Studente" altrimenti un "Docente". Tale rappresentazione tiene conto sia che "Docente" e "Studente" sono persone, e che possa essere caratterizzato il loro ruolo. Ammettere però tutti gli attributi sia di "Docente" che di "Studente", ciò potrebbe portare ad un tabellone ed un problema fisico.

Inserendo anche una relazione di insegnamento:



si avrà:

$\text{Insegnamenti}(\underline{\text{codice}}, \text{nome}, \text{docente})$

Dove docente è chiave esterna e rappresenta un codice fiscale:

$\text{Insegnamenti}[\text{docente}] \sqsubseteq_{FK} \text{Persona}[\text{CF}]$

Non è detto però che quel codice fiscale a cui si fa riferimento appartenga ad uno "Studente".

Un'idea per risolvere il problema è il seguente:

Persona(CF, nome)  
 Studente(persona, S1, S2)  
 Docente(persona, SSD)  
 Insegnamento(codice, nome, docente)  
 $\text{Studente}[\text{persona}] \sqsubseteq_{FK} \text{Persona}[\text{CF}]$   
 $\text{Docente}[\text{persona}] \sqsubseteq_{FK} \text{Persona}[\text{CF}]$   
 $\text{Insegnamento}[\text{docente}] \sqsubseteq_{FK} \text{Docente}[\text{persona}]$

Non si crea né totalità (possono esserci persone che non sono né studenti né docenti) né esclusività (Studenti e Docenti sono sottoinsiemi di Persona, ma non è detto che siano intersecati).

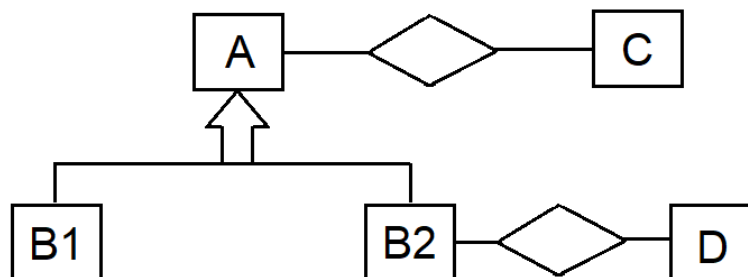
Un'altra idea potrebbe essere quella di non materializzare "Persona":

Studenti(CF, nome, S1, S2)  
 Docenti(CF, nome, SSD)  
 Insegnamenti(codice, nome, docente)

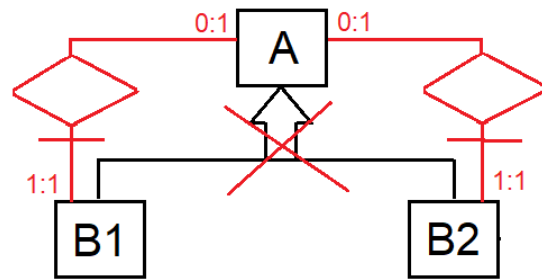
in questa configurazione, viene solo a perdersi l'esclusività.

In realtà NON ESISTE un modo senza alcun problema per rappresentare a livello logico una generalizzazione.

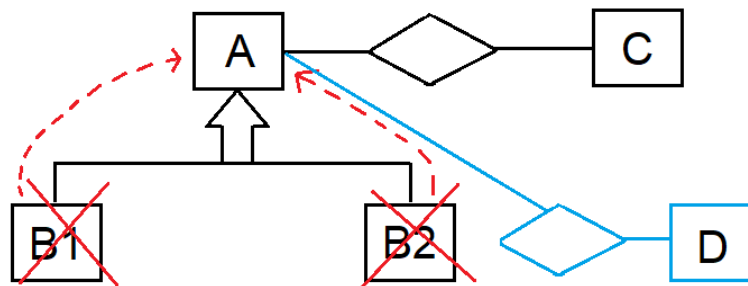
Considerato lo schema generalizzato:



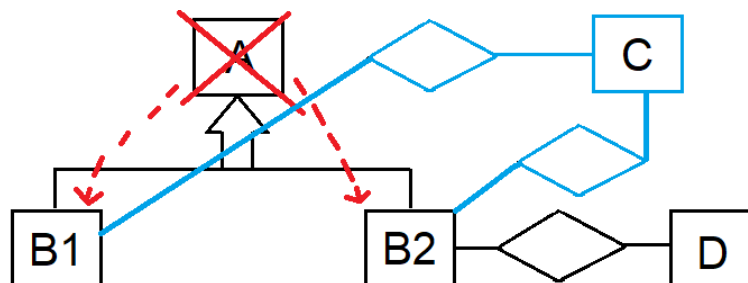
Ogni volta si va a creare uno **SCHEMA INTERMEDIO** che elimina la generalizzazione, inserendo due relazioni, e perdendo esclusività e totalità, ma vengono distinti B1 e B2:



Un'altra possibilità è quella di cancellare B1 B2 ed accorpare gli attributi in A, spostando naturalmente la relazione:



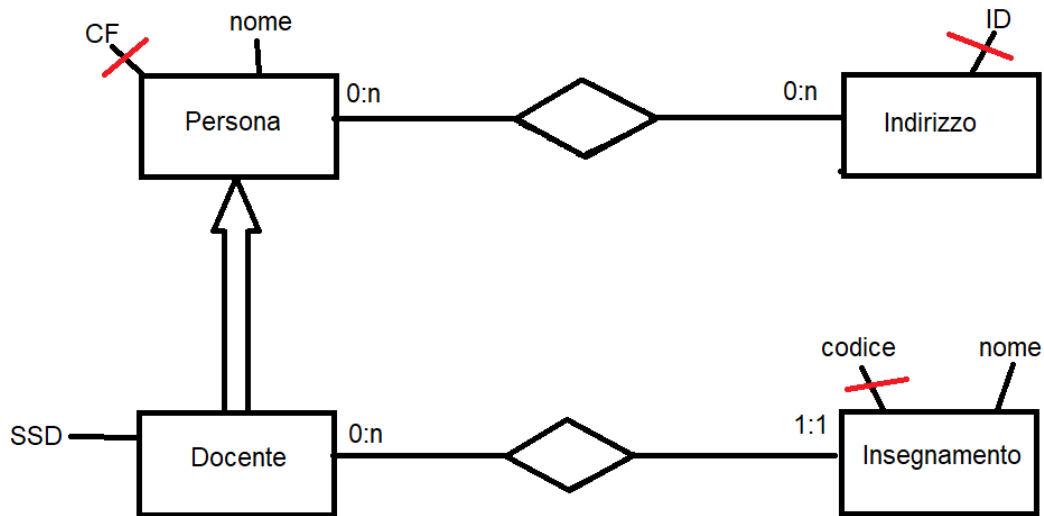
Infine si può scegliere di non materializzare la A, copiando tutti i suoi attributi in B1 e B2 e creando nuove relazioni tra C-B1 e C-B2:



La scelta di una di queste possibili soluzioni, è data dal particolare caso in analisi.

#### 4.4 ISA nel modello logico

Dato il modello E/R:



Nell'ISA non si ha totalità ed esclusività, poiché può essere tradotta facilmente:

Persona(CF, nome)  
Docente(persona, SSD)  
Insegnamento(codice, nome, docente)  
 $R_{PI}(\underline{persona}, \underline{indirizzo})$

Se non si avesse "Insegnamento" potrebbe essere più comodo avere:

Persona(CF, nome, docente(SI/NO), SSD)

Dove SSD viene considerato solo nel momento in cui la persona è un docente.

## 5 Interrogazione sulle relazioni

Presa per buona la capacità di creare un database, è fondamentale apprendere come interrogare le tabelle del database. Le interrogazioni sulle relazioni prendono il nome di **QUERY**.

Ciò avviene con l'algebra relazionale che si occupa di manipolare la tabella per prelevare dei dati.

Supponiamo di avere una tabella "Fornitore" e una tabella "Merce" popolata:

PIVA	nome	città
P1	A	CS
P2	B	RE

codice	nome	marca
M1	X	alpha
M2	Y	beta

Le tabelle popolate con le tuple prendono il nome di **istanze della relazione**, mentre la forma per dichiarare una tabella è detta **forma della relazione**

### 5.1 Selezione

Prende per argomento una tabella e una o più condizioni che le tuple devono rispettare per essere restituite.

$$\sigma_{cond}(Tabella)$$

un esempio potrebbe essere  $\sigma_{citta='CS'}(Fornitore)$

... equivale ad un FOR "logico" sulle righe di "Fornitore". L'output è una tabella con la stessa struttura di "Fornitore":

PIVA	nome	città
P1	A	CS

Si parla di FOR logico in quanto non si sa come la struttura è implementata in memoria secondaria, si può paragonare ad un Iteratore su cui viene fatta una next.

La condizione su  $\sigma$  può essere anche combinata utilizzando l'algebra booleana:

$$\sigma_{(citta='CS')OR(PIVA \neq 'P1' AND nome \neq 'C')}(Fornitore)$$

Tutti gli operatori dell'algebra relazione sono detti **RIENTRANTI**, ovvero, se applicato ad un dominio, restituisce oggetti dello stesso dominio.

$\sigma$  come gli altri operatori può essere concatenata, ad esempio:

$$\begin{aligned} &\sigma_{citta='CS'}(\sigma_{nome='A'}(Fornitore)) \\ &\quad \Updownarrow \\ &\sigma_{(citta='CS' AND nome='A')}(Fornitore) \end{aligned}$$

Le due sintassi NON differiscono come efficienza, anche se potrebbe sembrare che utilizzando due volte  $\sigma$  la tabella fornitore debba essere scandita due volte a differenza della singola ricerca.

## 5.2 Proiezione

Alcune volte è necessario però filtrare le colonne, ad esempio se viene richiesto "l'elenco dei nomi dei fornitori".

Questa volta non vengono scelte le tuple da visualizzare ma le colonne.

Questa operazione si indica con  $\pi$  ed è detta Proiezione.

$$\Pi_{(listacolonne)}(Tabella)$$

ad esempio:

nome
A
B

Quando si parla di relazioni e interrogazioni su esse si identifica:

- ARITA  $\rightarrow$  numero di colonne(modificato da  $\Pi$ )
- CARDINALITA  $\rightarrow$  numero di righe(modificato da  $\sigma$ )

Tutti gli operatori sono rientranti, e possono essere anche innestati tra loro.

Ad esempio, si vogliono scegliere i "nomi dei fornitori di Reggio Calabria":

$$\Pi_{nome}[\sigma_{citta=RC}(Fornitore)]$$

↓

nome
B

Un altro esempio può essere quello di determinare le "Marche delle merci il cui nome è X"

$$\Pi_{marca}[\sigma_{nome='X'}(Marca)]$$

Inoltre, non sempre vengono indicati i nomi degli attributi, prediligendo l'enumerazione delle colonne:

\$1	\$2	\$3
codice	nome	marca

$$\Pi_{\$1}[\sigma_{\$2=X'}(Marca)]$$

Si suppone di voler conoscere i nomi dei fornitori e delle merci, uguali. Per cui è necessario interagire con entrambe le tabelle:

$$\Pi_{\$2}(Fornitori) \cap \Pi_{\$2}(Marca)$$

Quando si applica un operatore binario, come intersezione o unione i due argomenti devono essere comparabili, affinché ciò avvenga devono avere stessa arità ed i nomi degli attributi devono essere uguali, in caso contrario devono essere rinominati.

si nota che l'operatore  $\rho$ , che serve a **Rinominare** le colonne, fa riferimento alla colonna 1 , in quanto  $\Pi$  restituisce due tabelle a singola colonna.

Stesse proprietà si applicano all'unione.

Se si volessero invece conoscere i nomi di fornitori, presenti in "Fornitore" ma non in "Merce", si usa l'operatore di **Negazione**:

$$\Pi_{\$2}(Fornitore) \setminus \Pi_{\$2}(Merce)$$



### 5.3 Prodotto Cartesiano

Anche il prodotto cartesiano può essere utilizzato in algebra relazionale e serve a creare accoppiamenti, si ipotizzi di voler realizzare:

$$\text{Fornitore} \times \text{Marca}$$

viene restituita un'unica tabella la cui struttura è una concatenazione tra più strutture:

PIVA	nome	città	codice	nome	marca

La notazione posizionale si attua anche perché possono nascere attributi, con stesso nome, nella stessa tabella. La tabella sarà così popolata:

PIVA	nome	città	codice	nome	marca
P1			M1		
P1			M2		
P2			M1		
P2			M2		
P3			M1		
P3			M2		

Si ha una concatenazione tra le tuple di tutte e due le relazioni

#### ESEMPI

- Conoscere le coppie  $\langle \text{PIVA}, \text{nome} \rangle$  dei fornitori

$$\Pi_{PIVA, nome}(Fornitore)$$

- Coppie  $\langle \text{PIVA}, \text{nome} \rangle$  dei fornitori che stanno in città diverse da RC

$$\Pi_{PIVA, nome}(\sigma_{città \neq RC}(Fornitore))$$

Il costo della proiezione data una tabella è  $O(n^2)$ , ciò deriva dal fatto che per ogni tupla va verificata se questa già esiste nella tabella delle soluzioni, per cui si hanno un numero di operazioni:

1° passo  $\rightarrow 1$

2° passo  $\rightarrow 2$

3° passo  $\rightarrow 3$

.

.

questo si generalizza nella formula di Gauss  $\frac{n(n+1)}{2}$ , da cui deriva dunque la complessità.

Questa query poteva essere anche scritta come:

$$\Pi_{PIVA, nome}(Fornitore \setminus \sigma_{citta \neq RC'}(Fornitore))$$

- Codice delle merci che si chiamano 'X' o che sono prodotte dalla marca 'gamma'

$$\Pi_{\$1}[\sigma_{(\$2=X) \vee (\$3=gamma)}(Merce)]$$

- Tutte le possibili coppie di nomi che rappresentano fornitori

$$\Pi_{\$2}(Fornitore) \times \Pi_{\$2}(Fornitore)$$

Il risultato ha cardinalità pari al quadrato delle tuple della tabella "Fornitore".

Un altro modo di realizzare la query è il seguente:

$$\Pi_{\$2, \$5}(Fornitore \times Fornitore)$$

per capire meglio il suo funzionamento, si ricorda che il prodotto cartesiano tra le due tabelle fornisce la seguente tabella:

PIVA	nome	città	codice	nome	marca

per cui si nell'utilizzo della proiezione si deve far riferimento alle colonne 2 e 5.

- Coppie di nomi di fornitori che stanno nella stessa città

$$\Pi_{\$2, \$5}(\sigma_{(\$3=\$5) \wedge (\$1 \neq \$4)}(Fornitore \times Fornitore))$$

## 5.4 Join

É utilizzato al posto del prodotto cartesiane, produce le combinazioni di tuple tra due tabelle tale che venga rispettata una determinata condizione.

$$Fornitore \bowtie_{\$3=\$3} Fornitore$$

I riferimenti alle colonne, quando si usa questo operatore, si riferiscono alle singole tabelle e non ad una nuova tabella con  $\text{arità} = \text{arità}(\text{Fornitore}) * 2$

In conclusione la Join equivale ad un prodotto cartesiano sulla quale è applicata una selezione sopra, si chiama infatti **OPERATORE DERIVATO**.

## ESEMPI

- Elenco dei codici delle merci fornite dai fornitori con nome 'A'

La tabella "Fornitura" non contiene il nome del fornitore, per cui è necessario creare una relazione tra la tabella "Fornitore" e "Fornitura".

Fornitore  $\times$  Fornitura

Fornisce la seguente relazione:

PIVA	nome	merce	PIVA	nome	marca

da cui:

$$\Pi_{\$5}(\sigma_{(\$1=\$4) \text{ and } (\$2='A')}(Fornitore \times Fornitura))$$

Un altro modo per definire la stessa Query consiste nel fare il prodotto cartesiano, con la tabella "Fornitore" caratterizzata da una selezione:

$$\sigma_{\$2='A'}(Fornitore) \times Fornitura$$

ancora una selezione per determinare le stesse PIVA tra le due tabelle:

$$\sigma_{\$1=\$4}(\sigma_{\$2='A'}(Fornitore) \times Fornitura)$$

e fare infine una proiezione sulla colonna delle merci:

$$\Pi_{\$5}(\sigma_{\$1=\$4}(\sigma_{\$2='A'}(Fornitore) \times Fornitura))$$

E ancora, la selezione sul prodotto cartesiano, è sostituibile con una join, per cui si ottiene:

$$\Pi_{\$5}[\sigma_{\$2='A'}(Fornitore) \bowtie_{\$1=\$1} (Fornitura)]$$

- Arricchire la tabella di "Fornitura" con i dettagli del fornitore corrispondente:

$$\Pi_{\$5='A'}(Fornitura \bowtie_{\$1=\$1} Fornitore)$$

- $\langle \text{nomeFornitore}, \text{Merce} \rangle$  tale che il fornitore fornisce la merce

$$\Pi_{\$5,\$8}((Fornitura \bowtie_{\$1=\$1} Fornitore) \bowtie_{\$2=\$1} Merce)$$

- Conoscere le merci fornite dai fornitori con città='CS'

$$\Pi_{\$8}\{[\sigma_{\$3='CS'}(Fornitore) \bowtie_{\$1=\$1} Fornitura] \bowtie_{\$5=\$1} Merce\}$$

- Nomi delle merci prodotte dalla marca alpha

$$\Pi_{\$2} [\sigma_{\$3='alpha'}(Merce)]$$

- Codici delle merci fornite da almeno un fornitore a meno di 10 euro

- Nomi delle merci prodotte a Milano

- Nomi delle merci fornite a più di 20 euro dai fornitori di Catanzaro

- Coppie di città tali che la prima delle due città fornisce almeno una merce prodotta nella seconda città della coppia.

- Nomi dei fornitori che forniscono almeno due merci

## 5.5 Differenze tra Join e Sottrazione

La Join scandisce la tabella e se trova una tupla che soddisfa la condizione, la restituisce. L'operatore di Sottrazione, invece prende l'elemento desiderato e scandisce la tabella, lo restituisce solo se non è presente in nessuna riga di una specifica tabella.

A livello logico la Join così come la Selezione, traducono il **Quantificatore Esistenziale**( $\exists$ ), mentre la sottrazione traduce il **For all**( $\forall$ ).

Entrambi lavorano come fossero DUE For innestati.

## 5.6 Interrogazione particolari

### Fornitori che producono almeno tre merci

La ricorrenza che segue "almeno" indica quante volte è necessario la join della Tabella su cui si sta lavorando, con se stessa:

$$(Fornitura \bowtie_{\$1=\$1 \ \& \ \$2 \neq \$2} Fornitura) \bowtie_{\$1=\$2 \ \& \ \$2 \neq \$2 \ \& \ \$5 \neq \$2} Fornitura$$

### Nomi dei fornitori che forniscono esattamente 1 merce

Per realizzare questa query si ragiona sul fatto che:

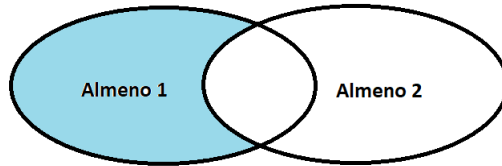
- I fornitori che forniscono esattamente una merce fanno parte dell'insieme dei fornitori che forniscono almeno una merce:

$$Almeno1 = \Pi_{\$1}(Fornitura)$$

- I fornitori che forniscono esattamente una merce non fanno parte dei fornitori che ne forniscono almeno due:

$$Almeno2 = \Pi_{\$1}(Fornitura \bowtie_{\$1=\$1 \ \& \ \$2 \neq \$2} Fornitura)$$

Ragionando in termini insiemistici, si identifica la seguente configurazione:



Per cui:

$$Esattamente1 = Almeno1 - Almeno2$$

La query finale sarà la seguente:

$$\Pi_{\$3}(Esattamente1 \bowtie_{\$1=\$1} Fornitore)$$

### Nomi di fornitori che forniscono al più due merci

Si può ragionare in modo simile al precedente. ovvero escludendo tutti quei fornitori che forniscono almeno 3 merci:

$$AlPiu2 = \Pi_{\$1}(Fornitore) - Almeno3$$

I risultati intermedi delle query si chiamo **VISTE**.

### **Fornitori che non forniscono merci**

Presi tutti i fornitori si eliminano quelli che forniscono almeno una merce:

$$\Pi_{\$1}(Fornitore) - \Pi_{\$1}(Fornitura)$$

### **Prezzo minimo a cui viene fornita la merce M1**

Si ragiona in maniera inversa, partendo dall'individuare i prezzi per la merce M1 che sicuramente non sono minimi. Un prezzo è NON minimo quando sicuramente ne esiste almeno uno più piccolo:

$$PrezziNonMinimi = \Pi_{\$3} [\Pi_{\$2=M1}(Fornitura) \bowtie_{\$2=\$2\$3>\$3} Fornitura]$$

Basterà quindi escludere queste tuple da tutti i prezzi di M1, per ottenere il prezzo minore:

$$PrezzoMinimo = \Pi_{\$3} [\sigma_{\$2=M1}(Fornitura)] - PrezziNonMinimi$$

### **Nomi dei fornitori che forniscono tutte le merci**

Si selezionano i fornitori tali per cui esiste una merce che essi non forniscono:

$$FornituraNO = \Pi_{\$1}(Fornitore) \times \Pi_{\$1}(Merce) - \Pi_{\$1,\$2}(Fornitura)$$

A questo punto basterà sottrarre il risultato appena ottenuto a tutti i fornitori:

$$\Pi_{\$3} \{ [\Pi_{\$1}(Fornitura) - \Pi_{\$1}(FornituraNO)] \bowtie_{\$1=\$1} Fornitore \}$$

### **Marche che producono almeno due merci, ciascuna delle quali è fornita da almeno due fornitori**

Si identificano le marche che producono almeno due merci, utilizzando la Join:

$$Marche2Merci = \Pi_{\$3}(Merce \bowtie_{\$3=\$3\$1 \neq \$1} Merce)$$

è necessario anche individuare anche le merci fornite da due fornitori:

$$Merci2Forn = \Pi_{\$2}(Fornitura \bowtie_{\$1 \neq \$1\$2=\$2} Fornitura)$$

Fatto questo, si aggiungono alla precedente query i dettagli delle merci:

$$Merci2FornDett = Merci2Forn \bowtie_{\$1=\$1} Merce$$

Ragionando come nella query "Marche2Merci" si prolunga la tabella ottenuta con un'altra istanza di se stessa:

$$\Pi_{\$4} [Merci2FornDett \bowtie_{\$4=\$4\$1 \neq \$1} Merci2FornDett]$$

## **Merci prodotte a Cosenza fornite da esattamente un fornitore**

Si scandiscono inizialmente le merci fornite esattamente da almeno due fornitori

$$Almeno2F = \Pi_{\$2}(Fornitura \bowtie_{\$1 \neq \$1 \& \$2 = \$2} Fornitura)$$

Per determinare le merci fornite da un fornitore, si prendono quelle in Fornitura togliendo quelle appena determinate:

$$Esattamente1 = \Pi_{\$2}(Fornitura) - Almeno2$$

A questo punto si prolungano le istanze con i dettagli della merce e della marca in modo da poter risalire alla sede di produzione:

$$\Pi_{\$1} \{ \sigma_{\$6 = 'CS'} [Esattamente1 \bowtie_{\$1 = \$1} Merce] \bowtie_{\$4 = \$1} Marca \}$$

## **Coppie (NM,P) dove NM è il nome di una merce prodotta a Cosenza e é è il prezzo massimo al quale la merce viene fornita a Reggio Calabria**

Si scompone il problema:

- Massimo prezzo al quale la merce è fornita a RC:

$$FornitureRC = \Pi_{\$1, \$2, \$3} [Fornitura \bowtie_{\$1 = \$1} (\sigma_{\$3 = 'RC'} Forniture)]$$

- Si identificano le coppie <Merce, Prezzo> che non presentano prezzo massimo:

$$coppieNonMax = \Pi_{\$2, \$3} FornitureRC \bowtie_{\$3 < \$3} FornitureRC$$

- Si identifica il prezzo massimo per ogni fornitura a RC:

$$coppiaMax = \Pi_{\$2, \$3} [FornitureRC - coppieNonMax]$$

A questo punto si considerano solo le coppie in cui la merce è prodotta a Cosenza

$$\Pi_{\$4, \$2} [(CoppiaMax \bowtie_{\$1 = \$1} Merce) \bowtie_{\$5 = \$1} Marca]$$

## **Nomi dei fornitori che fornisco tutte le loro merci a piu di 10 euro**

La query si può così riformulare:

”Nomi dei fornitori tali che non esiste alcuna merce da essa fornita a meno di 10 euro”

Si identificano i fornitori che forniscono merce a meno di 10 euro:

$$FornitoriNO = \Pi_{\$1} [\sigma_{\$3 < 10}(Fornitura)]$$

Si rimuovono dai fornitori, quelli appena identificati:

$$FornitoriSi = \Pi_{\$1}(Fornitore) - FornitoreNO$$

A questo punto si termina la query prendendo i nomi::

$$\Pi_{\$3}(FornitoriSi) \bowtie_{\$1=\$1} Fornitore$$

### **Merci non fornite a Reggio Calabria**

La query si può così riformulare:

”Nomi delle merci tali che non esiste alcun fornitore di Reggio Calabria che le fornisce”

Si considera la merce fornita dai fornitori di RC:

$$MerceRC = \Pi_{\$5} [\sigma_{\$3=RC}(Fornitore) \bowtie_{\$1=\$1} Fornitura]$$

A questo punto si eliminano le tuple appena determinate dalla lista delle merci fornite:

$$MerceNORC = \Pi_{\$1}(Merce) - MerceNO$$

Prolungando il tutto con Merce si estrae il nome:

$$\Pi_{\$3} [MerceNORC \bowtie_{\$1=\$1} Merce]$$

### **Merci fornite da tutti i fornitori**

La query si può così riformulare:

”Merci tali che non esistono fornitori che non le forniscono”

Si definiscono le merci tali che esistono fornitori che non le forniscono:

$$FornituraNO =_{\$1} (Merce) \times \Pi_{\$1}(Fornitori) - \Pi_{\$2,\$1}(Fornitura)$$

Si rimuove l’insieme delle merci appena trovato, dall’insieme di tutte le merci:

$$\Pi_{\$1}(Merce) - \Pi_{\$1}(FornituraNO)$$

### **Coppie F’ e F” che forniscono lo stesso insieme di merci**

Si riformuli la query:

”Coppie F’, F” di fornitori tali che non esiste alcuna merce fornita da F’ ma non da F” e viceversa”



Si cercano le coppie di fornitori tali che esiste una merce fornita dal primo ma non dal secondo:

$$\begin{aligned} FornituraNO &= \Pi_{\$1}(Fornitore) \times \Pi_{\$1}(Merce) - \Pi_{\$1,\$2}(Fornitura) \\ CoppieNO &= \Pi_{\$1,\$3}(Fornitura \bowtie_{\$2=\$2} FornituraNO) \end{aligned}$$

Si creano le coppie di fornitura a cui si sottraggono quelle appena trovate:

$$\Pi_{\$1}(Fornitore) \times \Pi_{\$1}(Fornitura) - CoppieNO] - \Pi_{\$1,\$2}(CoppieNO)$$

## 6 SQL

My SQL è un linguaggio che permette di interrogare i database. Si può dividere in due sottolinguaggi:

- DDL(DATA DEFINITION LANGUAGE) → è una serie di costrutti che vengono definiti dal linguaggio che consentono di:
  - **Creare** liste, tabelle, utenti
  - **Modificare** liste, tabelle, utenti
  - **Eliminare** liste, tabelle, utenti
- DML(DATA MANIPULATION LANGUAGE) → è una serie di costrutti che vengono definiti dal linguaggio che consentono di:
  - **Inserire** tuple
  - **Modificare** tuple
  - **Eliminare** tuple
  - **Estrarre** tuple

My SQL è un server quindi che gestisce i vari utenti e i loro dati tramite le query. Quando si vuole creare un database bisogna come prima cosa creare uno spazio logico

```
CREATE DATABASE MyDB;
```

```
USE MyDB;
```

Dopo la seconda istruzione tutti i comandi scritti nel codice finiranno in quello spazio logico. Per creare una tabella usiamo:

```
CREATE TABLE fornitore AS;  
(PIVA CHAR(20) PRIMARY KEY;  
NOME VARCHAR(40) UNIQUE;  
CITTA VARCHAR(20);  
);
```

Dove:

- CHAR(20) indica una sequenza di 20 caratteri
- VARCHAR(40) indica una sequenza di 40 caratteri di lunghezza variabile
- PRIMARY KEY indica che l'attributo è una chiave primaria
- UNIQUE indica una chiave non primaria

Se si volesse imporre che il valore di un attributo non possa essere *null*, è necessario aggiungere la scrittura NOT NULL.

Un modo alternativo per creare una tabella è:

```
CREATE TABLE fornitore AS;  
(PIVA CHAR(20);  
NOME VARCHAR(40) UNIQUE;  
CITTA VARCHAR(20);  
PRIMARY KEY PIVA;  
);
```

Si crea anche la tabella Merce e Fornitura:

```
CREATE TABLE Merce AS;  
(CODICE NUMBER(20) PRIMARY KEY;  
NOME VARCHAR(40) UNIQUE;  
MARCA VARCHAR(20);  
);  
CREATE TABLE Fornitura AS;  
(fornitore CHAR(20);  
merce NUMBER(40);  
PREZZO NUMBER(4,2);  
PRIMARY KEY fornitore  
REFERS TO fornitore(PIVA);  
PRIMARY KEY merce REFERS TO Merce(CODICE);  
);
```

Dove con NUMBER(40) si indica una sequenza di numeri mentre con NUMBER(4,2) indichiamo un numero formato da 4 numeri interi e due decimali. In fondo la tabella *fornitura* è riportata la sintassi per definire una *foreign key*. Se si volesse utilizzare un database già esistente bisogna usare la scrittura:

```
USE MyDB;
```

e di seguito vengono riportati i comandi per cancellare e modificare una tabella:

**DROP** tabella

**ALTER TABLE** tabella (  
modifiche  
);

Per popolare una tabella si individuano i seguenti comandi:

- Inserimento

**INSERT INTO** tabella **VALUES**  
(x,y,z ,... , );

- Eliminazione

**DELETE:** tupla da eliminare

- Aggiornamento

**UPDATE:** tupla da modificare

- Estrazione

**SELECT:** tupla da estrarre

L'argomento di **SELECT** può essere un'espressione qualunque:

**SELECT** PIVA,NOME;  
**FROM** Fornitore;

Per visualizzare l'intero contenuto di una tabella, si scrive:

**SELECT** \*;  
**FROM** tabella da visualizzare;

Se vogliamo ribattezzare l'attributo di una tabella, si scrive:

**SELECT** attributo **AS** attributoRinominato;  
**FROM** tabella;

Con più attributi il **SELECT AS** va assegnato singolarmente.

Si supponga ora di determinare:

”Nomi dei fornitori che stanno a Cosenza oppure quelli la cui partita IVA è 'F1' e il nome è diverso da 'A'”:

$$\Pi_{\$2} [\sigma_{\$3='CS' \vee \$1='F1' \wedge \$2 \neq 'A'}(Fornitore)]$$

```
SELECT nome;
FROM Fornitore;
WHERE CITTA='CS' OR PIVA='F1' AND NOME <> A
```

Si può quindi associare a  $\Pi$  il comando SELECT e a  $\sigma$  il comando WHERE

## 6.1 Join in MySQL

Per restituire le coppie di nomi di fornitori diversi dalla stessa città. Il algebra relazionale:

$$\Pi_{\$2, \$5}(Fornitore \bowtie_{\$1 \neq \$1 \wedge \$3 = \$3} Fornitore)$$

riscrivendo la *join* come combinazione di prodotto cartesiano e *selezione*:

$$\Pi_{\$2, \$5} [\sigma_{\$1 \neq \$1 \wedge \$3 = \$3} (Fornitore \times Fornitore)]$$

La scrittura SQL risulta più diretta:

```
SELECT F'.nome, F'''.nome;
FROM Fornitore AS F', Fornitore AS F'';
WHERE F'.CITTA=F'''.CITTA AND F'.PIVA <> F'''.PIVA;
```

## 6.2 Operazioni Algebra relazionale

Sottrazione → EXCEPT

Unione → UNION

Intersezione → INTERSECTION

Il prodotto cartesiano si ottiene inserendo più tabelle alla volta dopo il comando FROM.

### 6.3 Creazione Liste

```
CREATE VIEW Almeno1(Fornitore) AS
```

```
(  
    SELECT fornitore;  
    FROM Fornitura;  
);
```

```
CREATE VIEW Almeno2(Fornitore) AS
```

```
(  
    SELECT x1.fornitore;  
    FROM Fornitura AS x1, Fornitura AS x2;  
    WHERE x1.fornitore=x2.fornitore AND x1.merced > x2.merced;  
);
```

Le liste servono in quanto non è possibile implementare query temporanee. Quindi create le due liste, per sapere i fornitori che producono necessariamente una merce è necessario fare:

```
SELECT *;  
FROM Almeno1
```

```
EXCEPT
```

```
SELECT *;  
FROM Almeno 1
```

### 6.4 Interrogazioni

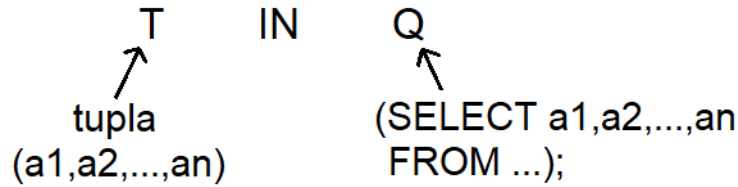
I predicati:

In	NOT IN	EXIST	NOT EXIST
----	--------	-------	-----------

traducono l'appartenenza o meno di un elemento in una tabella e l'esistenza o meno di un elemento. Di fatto anche la Join determina una relazione esistenziale ma produce anche un allungamento della tabella.

### 6.4.1 IN

La sintassi SQL dell'operatore IN è la seguente:



Si determinano i nomi dei fornitori che forniscono almeno una merce:

$$\begin{aligned} & \Pi_{\$2}(Fornitore \bowtie_{\$1=\$1} Fornitura) \\ & \quad \downarrow \\ & \Pi_{\$2}[\sigma_{\$1=\$4}(Fornitore \times Fornitura)] \end{aligned}$$

che in SQL si traduce in:

```
SELECT F.nome;
FROM Fornitore F, Fornitura X;
WHERE F.PIVA=X.fornitore;
```

Il predicato IN serve a stabilire se un valore di un attributo o una combinazione di valori di più attributi, fa parte di una tabella riportata a destra nell'algebra relazionale:

```
SELECT F.nome;
FROM Fornitore F;
WHERE F.PIVA In (SELECT fornitore
                  FROM Fornitura);
```

es.

”Coppie <nomeF,nomeM> tali che il primo fornisce la seconda”

$$\Pi_{\$2,\$8}[(Fornitore \bowtie_{\$1=\$1} Fornitura) \bowtie_{\$5=\$1} Merce]$$

che in SQL si traduce in:

```
SELECT F.nome,M.nome;
FROM Fornitore F, Fornitura X, Merce M;
WHERE F.PIVA=X.fornitore AND M.codice=X.merce;
```

Si ricorda che i test di uguaglianza possono essere scritti anche invertendoli.

Il codice SQL può essere riformulato utilizzando un test di appartenenza. Riformulando la query:

”Coppie di nomi di fornitori e merci tali che le corrispondenti coppie (PIVA,codice) appartengono a fornitore”

```
SELECT F.nome, M.nome;
FROM Fornitore F, Merce M;
WHERE (F.PIVA,M.codice) IN (SELECT fornitore, merce
FROM Fornitura);
```

es.

”coppia di <nomeF,merce> tali che il fornitore non fornisce la merce”

$$\underbrace{\Pi_{\$1,\$4}(Fornitori \times Merce) - \Pi_{\$1,\$2}(Fornitura)}_A$$

Per ottenere le coppie di nomi è necessario prolungare A con fornitore e merce:

$$\Pi_{\$4,\$7}(A \bowtie_{\$1=\$1} Fornitore) \bowtie_{\$2=\$1} Merce)$$

In SQL:

```
CREATE VIEW CoppieCodicino (F.nome,M.nome) AS
(SELECT F.PIVA,M.codice;
FROM Fornitore F,merce M;

EXCEPT

SELECT Fornitore,Merce;
FROM Fornitura;
)
```

```
SELECT F.nome,M.nome;
FROM CoppieCodicino C, Fornitore F, Merce M;
WHERE C.fornitore=F.PIVA AND C.merce=M.codice
```



es.

”Coppie di nomi di fornitori e merci tali che le coppie (PIVA, codice) corrispondenti non appartengono a fornitore”

```
SELECT F.nome, M.nome;  
FROM Fornitore F, Merce M;  
WHERE (F.PIVA,M.codice) NOT IN (SELECT fornitore ,merce  
                                FROM Fornitura)
```

#### 6.4.2 Linguaggio Dichiarativo

Si noti che il linguaggio SQL è un linguaggio che consente di esprimere le query in forma dichiarativa. Un linguaggio è DICHIARATIVO quando consente di esprimere un concetto dichiarandone le proprietà.

L'algebra relazionale invece è procedurale, nello scrivere una query, questa non è indice immediato del significato della proprietà caratteristica, ma è una spiegazione di una procedura da seguire per ottenere il risultato.

Alcuni motori SQL non supportano IN e NOT IN in cui a sinistra ci sono tuple, ma supportano solo il caso in cui a sinistra si ha un solo attributo.

#### 6.4.3 EXIST

L'EXIST così come il NOT EXIST a sinistra, ha un solo parametro. Tale parametro una select:  
es.

”Nomi di forniture che forniscono almeno 1 merce”

```
SELECT F.nome;  
FROM Fornitore F;  
WHERE EXISTS (SELECT *;  
              FROM Fornitore X;  
              WHERE X.fornitore=F.PIVA);
```

per ogni riga di Fornitore verifica che il risultato della query sotto è NON VUOTO. La query sotto seleziona l'insieme delle forniture tale che:

X.fornitore=F.PIVA

dove F.PIVA corrisponde alla PIVA appartenente alla riga di Fornitore selezionata.  
la WHERE corrisponde a scrivere:

...  
**WHERE** F.PIVA **IN** (**SELECT** fornitore  
                  **FROM** Fornitore );

es.

”Determinare, per ogni merce, il prezzo minimo di fornitura”

↓

”Per ogni merce, il prezzo tale che non ne esiste uno più piccolo”

**SELECT** X1.merce ,X1.prezzo ;  
**FROM** Fornitura XI;  
**WHERE NOT EXISTS** (**SELECT** \*  
                  **FROM** FornituraX2 ;  
                  **WHERE** X2.merce=X1.merce  
                  **AND** X2.prezzo < X1.prezzo );

es.

”Nomi dei fornitori che forniscono tutte le merci”

↓

”Nomi dei fornitori tali che non esistono merci M, tali che F,M non è in Fornitura”

Che in algebra relazionale:

$$FornituraNO = [\Pi_{\$1,\$4}(Fornitore \times Merce)] - \Pi_{\$1,\$2}(Fornitura)$$

essendo richieste le coppie di nomi è necessario prolungare la tabella con fornitore e merce:

$$\Pi_{\$1} [\Pi_{\$1}(Forniture) - \Pi_{\$1}(FornitoriNO)] \bowtie_{\$1=\$1} (Forniture)$$

in SQL si ottiene:

```
SELECT F.nome;  
FROM Fornitore F;  
WHERE NOT EXISTS (SELECT *;  
                   FROM Merce M;  
                   WHERE (F.PIVA,M.merce) NOT IN (SELECT fornitore ,merce;  
                                                FROM Fornitura );  
                   )
```

es.

”Coppie di nomi F1, nome F2 tali che l’insieme delle merce formate da F1 è uguale all’insieme delle merci formate da F2”

↓

”Non esiste alcuna merce formata da F1 ma non da F2 e viceversa”

```
SELECT F1.nome,F2.nome;  
FROM Fornitore F1, Fornitore F2;  
WHERE NOT EXISTS (SELECT *;  
                   FROM Fornitura X1;  
                   WHERE X1.Fornitore=F1.PIVA AND  
                   (F2.PIVA,X1.merce) NOT IN  
                   (SELECT fornitore ,merce;  
                   FROM Fornitura );  
                   )  
AND NOT EXISTS (SELECT *;  
                   FROM Fornitura X;  
                   WHERE X.fornitore=F2.PIVA  
                   AND (F1.PIVA,X.merce) NOT IN  
                   (SELECT fornitore , merce;  
                   FROM Fornitura );  
                   )  
AND F1.nome !=F2.nome;
```

#### 6.4.4 Aggregazione

I costrutti di aggregazione sono dei possibili calcoli eseguibili sulle tuple di una tabella e che consentono di usare un valore sintetico di rappresentazione di tante tuple.

Dato un insieme di tuple che rappresentano clienti caratterizzati da un'età, un'aggregazione potrebbe essere calcolare l'età media e il max/min.

In SQL una prima forma di calcolo consiste nell'unire attributi della stessa TUPLA. Si considerino :

Fattura(numero,anno, cliente, data)  
Composizione(numeroF,annoF,prodotto,qta,prezzoU)

la tabella di composizione sarà tipo:

si noti che in questa non è espressamente indicato quanto è stato speso in totale.

Se si volesse rappresentare l'informazione, quando viene scritta la SELECT è possibile richiamare delle funzioni che lavorano sul singolo attributo o che mettono insieme più attributi:

```
SELECT numeroF , annoF , prodotto , qta*prezzo U prezzoTOT;  
FROM Composizione;
```

ottenendo:

Possono essere effettuate numerose operazioni e le operazioni principali che si utilizzeranno sono:

SUM            COUNT            MIN            MAX            AVG

L'operatore di aggregazione viene eseguito per ogni gruppo di tuple che ha superato la condizione di where.

es.

Cliente(CF,nome,citta,eta)

```
SELECT max( et ) AS et Massima;  
FROM Cliente;
```

es.

```
SELECT max( et ) AS et Massima;  
FROM Cliente;
```

prende la tabella cliente e restituisce il numero di CF

Se si fossero voluti i clienti di Cosenza, sarebbe stato necessario filtrare le tuple con:

```
WHERE citta='CS';
```

Per conoscere il numero di righe di una tabella, piuttosto che specificare un'attributo, al posto di questo si mette un \*.

Si noti che una query del tipo:

```
SELECT COUNT( Citta );  
FROM Cliente ;
```

Non restituisce il numero di diversi valori dell'attributo città, ma il numero di volte in cui l'attributo città è diverso da null.

Per ottenere il numero di diverse occorrenze di un attributo si usa la scrittura SQL:

```
SELECT COUNT -DISTINCT( citta )  
FROM Cliente ;
```

es.

"Si supponga di voler conoscere il numero di clienti di RC"

```
SELECT COUNT(*);  
FROM Cliente ;  
WHERE citta='RC';
```

ma avendo n città e volendo sapere per ognuna il numero di clienti con residenza risulta scomodo scrivere n query.

Per questo motivo è introdotto il costrutto Group By

#### 6.4.5 Group By

```
SELECT citta ,COUNT(*);  
FROM Cliente ;  
WHERE nome != 'Pasquale' ;  
GROUP BY citta ;
```

Dalla tabella cliente, si filtrano le tuple in base al nome e viene prodotta una tabella per ogni diversa città arricchita dalle tuple filtrate

La GROUP BY, dato un insieme di tuple originarie, ne fa una partizione.

Tra loro non c'è intesezione e insieme costituiscono la totalità della tabella originale.

In base alla select vengono poi prodotte le sottotabelle:

Per ottenere la 2° colonna della tabella sarà necessaria la query:

```
SELECT COUNT(*);  
FROM Cliente;  
GROUP BY Citta;
```

è evidente che una tabella del genere non dà alcuna informazione.

Si noti che una query del tipo:

```
SELECT citta , nome, COUNT(*);  
FROM Cliente;  
GROUP BY citta;
```

è errata, in quanto, per ogni città viene restituita una tupla, ma ogni tupla non ha un nome, in quanto le tuple in uno stesso raggruppamento di città, hanno nomi diversi.

In generale, quando si scrive una query in cui sono presenti degli aggregati, non è possibile scrivere attributi che non fanno parte della "group by" (Regola Sintattica)

Un altro esempio errato è il seguente:

```
SELECT nome,SUM(eta );  
FROM Cliente;
```

es.

Si considerino i seguenti schemi di relazione:

Cliente(CF,nome,città,età)

Prodotto(codice,nome, marca)

Fattura(numero,anno,cliente,data)

Composizione(numeroF,annoF,prodotto,qta,prezzoU)

"Calcolare l'età media dei clienti che non stanno a RC"

```
SELECT AVG(età );  
FROM Cliente;  
WHERE città != 'RC';
```

”Calcolare il numero complessivo di esemplari del prodotto P1 venduti a clienti di CS”

$$[\sigma_{\$3='CS'}(Cliente) \bowtie_{\$1=\$3} Fattura] \bowtie_{\$5=\$1} (\sigma_{\$3=\$1}(Composizione))$$

A questo punto, sapere il numero di esemplari di P1 consiste alla somma di tutti i valori della colonna 'qta'. Che in SQL si traduce in:

```
SELECT SUM( qta )  
FROM Cliente C, Fattura F, COmposizione X;  
WHERE C. citta='CS' AND X. prodotto='P1 '  
      AND F. cliente=C.CF  
      AND F. numero=X. numeroF  
      AND F. anno=X  
GROUP BY citta ;
```

”Numero di prodotti distinti caratterizzati dalla coppia <numeroF,annoF>”

Si consideri la tabella :

dalla tabella T1 e T2 vengono restituiti gli attributi della select. Si sviluppa:

```
SELECT numeroF, annoF, count district (prodotto);  
FROM Composizione;  
GROUP BY numeroF, annoF;
```

É possibile utilizzare piu aggregati all'interno della select, ottenendo che siano compatibili con la GROUP BY:

```
SELECT ... count(prodotto),sum(qta)
```

non è pero possibile innestare piu aggregati:

```
sum(max(count(...)))
```

”Conoscere per ogni fattura, l'importo totale”

```
SELECT numeroF, annoF, sum( qta*prezzoU );  
FROM Composizione;  
GROUP BY numeroF, annoF;
```

questo non è un innesto di operatori di aggregazione. Un'alternativa poteva essere quella di utilizzare una lista:

```
CREATE VIEW A (...) AS  
(SELECT numeroF , annoF , qta*prezzoU AS prezzoTproduzione );  
FROM Composizione;
```

```
SELECT numeroF , annoF , sum(prezzoTprodotto)  
FROM A;  
GROUP BY numreoF , annoF ;
```

La query può essere riscritta in maniera più diretta utilizzando il costrutto **HAVING**:

```
SELECT numeroF , annoF , SUM( qta*prezzo );  
FROM Composizione;  
GROUP BY numeroF , annoF ;  
HAVING COUNT( prezzo );
```



## 7 Dipendenza Funzionale

La dipendenza funzionale è un vincolo sul modello relazionale, è specificata su uno schema di relazione e dice quali istanze dello schema sono valide.

Si supponga di avere la relazione:

Cliente(CF, nome, eta, comune, CAP, prov)

con le seguenti istanze:

C1	A	20	Rende	87036	CS
C2	B	21	Cosenza	87100	CS
C3	C	18	ReggioC	87100	RC

Si nota la presenza di due tuple con stesso CAP, ma con comune diverso, questo nella realtà non è possibile, in quanto un CAP, è una porzione di un comune. Per cui i vincoli visti fino ad ora non sono esaustivi (chiave primaria, esterna).

La condizione è che se due tuple hanno lo stesso valore di CAP allora hanno lo stesso valore di comune. Un vincolo di questo tipo si chiama Dipendenza Funzionale e nella sua forma generale si indica con:

$$X \rightarrow Y$$

dove X e Y sono insiemi di attributi dello schema di relazione su cui è definita la D.F. , per cui:

$$CAP \rightarrow comune$$

ovvero il valore di CAP implica il valore del comune.

### DEFINIZIONE

Sia dato uno schema di relazione  $R(A_1, \dots, A_m)$ , Una dipendenza funzionale su tale schema è un'espressione del tipo:

$$X \rightarrow Y \quad (\text{con } X, Y \text{ attributi} \subseteq A_1, \dots, A_m)$$

Un'istanza r di R è consistente rispetto a  $X \rightarrow Y$  se, essendo  $X = \{X_1, \dots, X_n\}$ ,  $Y = \{Y_1, \dots, Y_k\}$ :

$$\begin{aligned} \forall t_1, t_2 \in r \quad & t_1[X_1] = t_2[X_1] \dots t_1[X_n] = t_2[X_n] \\ & \Downarrow \\ & t_1[Y_1] = t_2[Y_1] \dots t_1[Y_k] = t_2[Y_k] \end{aligned}$$

La scrittura:

$$\text{stato, categoria} \rightarrow \text{IVA}$$

afferma che, se due tuple, coincidono nella coppia  $\langle \text{stato, categoria} \rangle$  allora devono coincidere in IVA.

La dipendenza funzionale è simile al vincolo di chiave. Si suppone la relazione:

$$\text{Esame}(\text{matr}, \text{nome}, \text{data}, \text{materia}, \text{docente}, \text{voto})$$

una papabile chiave è la coppia  $\langle \text{matc}, \text{materia} \rangle$ , è possibile esprimere questa chiave mediante una dipendenza funzionale:

$$\text{matr}, \text{materia} \rightarrow \text{nome}, \text{data}, \text{docente}, \text{voto}$$

ovvero, un istanza di esame è consistente se prese due tuple, se queste coincidono in matricola e materia allora devono coincidere in tutti gli attributi.

Si suppongono di avere le istanze di Esame:

M1	A	BD	F	01/03/2021	30
M2	B	ASD	F'	02/03/2021	24
M1	C	ASD	F'	02/03/2021	25

queste rispettano la dipendenza funzionale definita, ma presentano comunque un errore, in quanto nella 1° tupla 'M1' ha come nome 'A', mentre nella 3° 'C'. Non è però pensabile di sostituire 'materia' con 'nome', per cui è necessaria un'altra dipendenza funzionale che non è rappresentabile con una chiave.

$$\text{matr} \rightarrow \text{nome}$$

La conclusione è che dato un schema di relazione, è possibile associare un'insieme di dipendenze funzionali  $F$ .

In SQL non c'è alcun costrutto nativo che permette di definire una dipendenza funzionale, ma è possibile inserire del codice. In particolare esiste il costrutto **CREATE TRIGGER . . . ON INSERT** ovvero una procedura eseguita ogni volta che è effettuato un inserimento e che permette di non accettare una tupla se non rispetta particolari condizioni.

Le dipendenze funzionali servono anche a verificare l'efficacia dello schema di una relazione, infatti, se nella parte sinistra di una dipendenza funzionale non si ha una chiave, questo implica la presenza di ridondanza nella tabella. Inoltre implicitamente ne fornisce anche la soluzione; nell'esempio precedente sarà necessario eliminare dallo schema l'attributo "nome", con la conseguente eliminazione di ridondanza, sarà poi messa a parte lo schema relazionale:

Studente(matr, nome)

caratterizzata da una sola dipendenza funzionale:

$$\text{matr} \rightarrow \text{nome}$$

che dimostra che non è presente ridondanza.

### DEFINIZIONE

Dato uno schema di relazione  $R(A_1, \dots, A_n)$  ed uno schema di relazione  $F$ , si dice che  $R$  in forma normale di BOYCE-CODD (BCNF), se:

$\forall X \rightarrow Y$ , l'insieme  $X$  è una **SUPERCHIAVE**, ovvero contiene al suo interno una chiave.

Dato uno schema di relazione:

$$R(A, B, C, D, E)$$

caratterizzato dalle seguenti dipendenze funzionali:

$$\left\{ \begin{array}{l} AB \rightarrow C \\ CD \rightarrow E \\ E \rightarrow AB \end{array} \right. \quad (1)$$

da cui si concludo logicamente che:

$$CD \rightarrow ABE$$

è stata trovata una dipendenza funzionale IMPLICATA dalla precedente, affermando che se due tuple coincidono in  $CD$ , allora coincidono in tutti gli altri attributi. Si conclude quindi che  $CD$  è una chiave ed in particolare una SUPERCHIAVE, non si può ancora definire "candidata" in quanto non è detto che sia minimale.

Dall'insieme delle dipendenze funzionali, se due tuple coincidono in  $C$  o in  $D$  non coincidono in altri attributi, questo porta a dire che  $CD$  è una CHIAVE CANDIDATA.

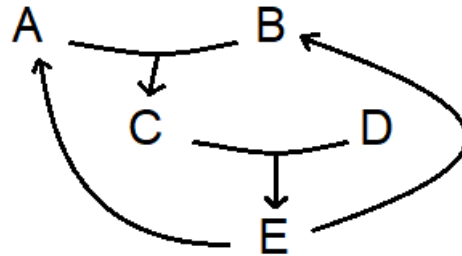
Il discorso è analogo per la coppia  $DE$ , di fatti  $E \rightarrow AB$ , ma  $AB \rightarrow C$ , per cui;

$$DE \rightarrow ABC$$

per cui  $DE$  è una chiave candidata; infine presa  $ABD$ ,  $AB \rightarrow C$  e  $AB \rightarrow E$ . L'insieme di chiavi candidate ottenuto è il seguente:

$$Keys\{CD, DE, ABD\}$$

Esiste un metodo meccanico per determinare le chiavi candidate, che fa uso del **GRAFO DELLE DIPENDENZE**. Si prendono le dipendenze e le si scrivono come nodi, a questo punto si crea per ogni dipendenza funzionale un arco multiplo che parte dai nodi a sinistra della dipendenza , arrivando ai nodi a destra:



il grafo permette di individuare facilmente tutte le chiavi candidate, ciò permette di scegliere quella primaria e inserire le altre come vincoli di unicità.

Una serie di implicazioni del tipo:

$$\left\{ \begin{array}{l} X \rightarrow Y \\ Y \rightarrow Z \\ X \rightarrow Z \end{array} \right. \quad (2)$$

presentano ridondanza, in quanto  $X \rightarrow Z$  è già implicato implicitamente.

Esistono anche forme di ridondanza che coinvolgono le parti sinistre delle dipendenze funzionali, ad esempio, si prendano:

$$\left\{ \begin{array}{l} A \rightarrow B \\ B \rightarrow C \\ AC \rightarrow D \end{array} \right. \quad (3)$$

Non ha senso scrivere  $AC \rightarrow D$ , in quanto essendo  $A \rightarrow B$  e  $B \rightarrow C$ , sicuramente date due tuple con stessa A, C sarà uguale. Per cui basterà:

$$A \rightarrow D$$

Facendo ancora riferimento allo schema relazionale precedente si identifica l'insieme di dipendenze funzionali F, caratterizzato dalle seguenti dipendenze funzionali:

$$\left\{ \begin{array}{l} AB \rightarrow C \\ CD \rightarrow E \\ E \rightarrow ABD \\ D \rightarrow B \\ EC \rightarrow D \end{array} \right. \quad (4)$$

Ci sono due possibili forme di ridondanza, si studia se è possibile ridurre le parti sinistre. Si analizzano le dipendenze funzionali:

- $AB \rightarrow C$

Si noti che la chiusura di A comprende solo A:

$$A^+ = \{A\}$$

In quanto se due tuple coincidono in A non coincidono in nient'altro. Stesso discorso per B. Per cui la parte sinistra non può essere ridotta.

- $CD \rightarrow E$

$C^+ = \{C\}$  e  $D^+ = \{D, B\}$ . Per cui non può essere ridotta.

- $EC \rightarrow D$

$E^+ = \{EABDC\}$  per cui la dipendenza può essere scritta nella forma più compatta

$E \rightarrow D$ .

L'insieme F parziale è il seguente:

$$\left\{ \begin{array}{l} AB \rightarrow C \\ CD \rightarrow E \\ E \rightarrow ABD \\ D \rightarrow B \\ E \rightarrow D \end{array} \right. \quad (5)$$

Si passa alla riduzione della parte destra, si prendono le singole dipendenze e si analizzano tutte le altre, se si ottiene quella in analisi, questa è superflua, per cui può essere eliminata:

- $AB \rightarrow C$

dove  $(AB)^+ = \{AB\}$ , per cui non si può ridurre.

- $CD \rightarrow E$ , si ha il discorso analogo alla dipendenza precedente.

- $E \rightarrow ABD$ , si prendono le dipendenze in forma canonica:

- $E \rightarrow A$

non esiste nessun'altra dipendenza con A sulla destra, questo indica che non può essere cacciata.

- $E \rightarrow B$

si nota che  $E^+ = \{EADB\}$ , per cui è superflua in quanto si arriva a questa dipendenza funzionale anche in un altro modo:

$$\left\{ \begin{array}{l} AB \rightarrow C \\ CD \rightarrow E \\ E \rightarrow AD \\ D \rightarrow B \\ E \rightarrow D \end{array} \right. \quad (6)$$

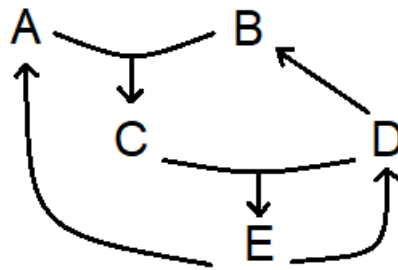
- $E \rightarrow D$

questa dipendenza è già presente singolarmente nell'insieme F, per cui se ne elimina una delle due:

$$\left\{ \begin{array}{l} AB \rightarrow C \\ CD \rightarrow E \\ E \rightarrow AD \\ D \rightarrow B \end{array} \right. \quad (7)$$

- $D \rightarrow B$ , si ha  $D^+ = \{D\}$

Si costruisce il grafo delle dipendenze per determinare le chiavi:



$$Keys = \{E, CD, AD\}$$

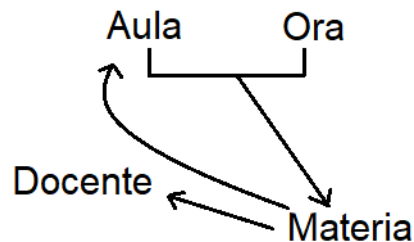
### Esempio

Orario(Aula, Ora, Materia, Docente)

Vale l'insieme di dipendenze funzionali:

$$\left\{ \begin{array}{l} \text{Materia} \rightarrow \text{Aula}, \text{Docente} \\ \text{Aula}, \text{Ora} \rightarrow \text{Materia} \end{array} \right. \quad (8)$$

Dalla quale si ottiene il grafo delle relazioni:



E si ricava l'insieme delle chiavi:

$$Keys = \{Aula - Ora, Ora - Materia\}$$

Non esistono altri insiemi minimali di attributi da cui dipende funzionalmente tutto.

Si usano per capire se la relazione è in BCNF o meno, ovvero se contiene ridondanze. Se per ogni dipendenza funzionale, la parte sinistra è una chiave, allora non sono presenti ridondanze. La dipendenza funzionale,  $Materia \rightarrow Aula, Docente$  presenta dunque ridondanza.

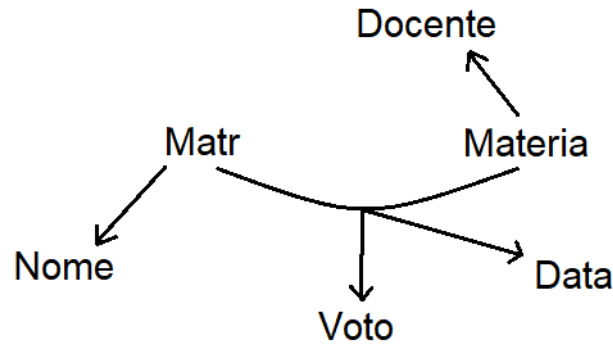
### Esempio

Si faccia riferimento allo schema relazionale e all'insieme F di dipendenze funzionali:

Esame(matr, nome, materia, docente, data, voto)

$$\left\{ \begin{array}{l} \text{Matr} \rightarrow \text{Nome} \\ \text{Materia} \rightarrow \text{Docente} \\ \text{Matr}, \text{Materia} \rightarrow \text{Data}, \text{Voto} \end{array} \right. \quad (9)$$





L'unica chiave candidata che si trova è:

$$Keys = \{Matr - Materia\}$$

Ispirandosi alle dipendenze, si creano delle relazioni separate:

$$\begin{array}{ll}
 \textit{Studente}(\underline{\textit{Matr}}, \textit{Nome}) & \textit{Matr} \rightarrow \textit{Nome} \\
 \textit{Insegnamento}(\underline{\textit{Materia}}, \textit{Docente}) & \textit{Materia} \rightarrow \textit{Docente}
 \end{array}$$

A questo punto la relazione esame sarà data da:

$$\textit{Esame}(\underline{\textit{Matr}}, \underline{\textit{Materia}}, \textit{data}, \textit{voto})$$

con la sola dipendenza funzionale  $\textit{Matr}, \textit{Materia} \rightarrow \textit{Data}, \textit{Voto}$

É stata fatta una DECOMPOSIZIONE IN BCNF.

Si parte da una relazione, per ottenere un'insieme di schemi sugli stessi attributi della relazione iniziale, tale che ciascuno degli schemi è in BCNF. Questa decomposizione presenta due proprietà:

- LOSSLESS JOIN

Un'istanza originaria di "Esame", si ottiene dalla Join di "Studente", "Insegnamento", "Esame"

- Tutti i vincoli di integrità definiti sullo schema iniziale, valgono anche sugli schemi separati, ovvero non si perdono dipendenze funzionali.

### Generalizzazione Algoritmo di Decomposizione in BCNF

Per quanto non sia sempre possibile fare una decomposizione in BCNF che sia senza perdita di informazione e senza perdita di dipendenza funzionale, almeno si riesce a garantire che non si perda l'informazione.

Preso una Relazione R con attributi  $A_1, \dots, A_n$ , e preso un insieme x:

$$X \subseteq \{A_1, \dots, A_n\}$$

ed un insieme Y:

$$Y \subseteq X^+ / X$$

dove  $X^+$  sono tutti gli attributi che dipendono funzionalmente da X. E infine con Z l'insieme rimanente:

$$Z = \{A_1, \dots, A_n\} / (X \cup Y)$$

Allora, preso:

$$R1(XY) \qquad R2(XZ)$$

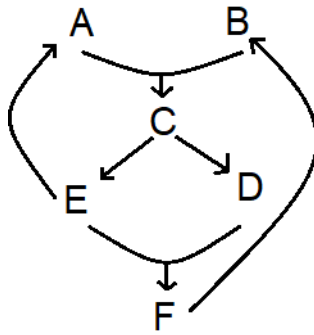
Si ottiene una decomposizione senza perdita di informazione, ovvero facendo la join tra le istanze degli schemi, si ottiene l'istanza originale.

Si verifica se le relazioni sono in BCNF, ed in caso contrario, si scompongono ulteriormente in maniera iterativa. Prima o poi il numero di attributi scende, in quanto viene diviso tra le relazioni.

Il caso limite è ottenere relazioni a due attributi, che sono sicuramente in BCNF.

Si supponga di avere una relazione R caratterizzata dal seguente schema funzionale:

$$R(ABCDEF) \qquad \left\{ \begin{array}{l} AB \rightarrow C \\ C \rightarrow ED \\ ED \rightarrow F \\ E \rightarrow A \\ F \rightarrow B \end{array} \right. \qquad (10)$$



Si determinano le chiavi:

$$Keys = \{AB, C, ED, EF, BE, AF\}$$

La relazione non è in BCNF, in quanto preso  $E \rightarrow A$  e  $F \rightarrow B$ , le parti sinistre non sono chiavi candidate.

Si applica l'algoritmo e si scorpora R, scegliendo un insieme X,Y e Z.

Come X si sceglie la parte sinistra di una dipendenza funzionale che non soddisfa la BCNF:

$$X=E$$

Come Y si sceglie invece la chiusura di X tranne X, per cui:

$$Y=A$$

Tutto il resto rimane in Z.

Si ottiene:

- $R1(AE)$

a cui si lega la dipendenza funzionale  $E \rightarrow A$ .

- $R2(BCDEF)$

a cui si legano le dipendenze funzionali  $C \rightarrow ED$ ,  $ED \rightarrow F$ ,  $F \rightarrow B$ .

Ci sono dipendenze funzionali non presenti in  $F$  in quanto non minimali, che si ottengono per proiezione. Fare le proiezioni significa scrivere tutte le dipendenze sugli attributi di un insieme implicate da quelle originali.

In questo non avere la A nell'insieme R2, rende non più implicata la dipendenza funzionale  $EB \rightarrow C$

R1, è caratterizzata da una sola dipendenza funzionale  $E \rightarrow A$ , e l'insieme delle chiavi è  $Keys = \{E\}$ , per cui la relazione è BCNF

R2 invece presenta le seguenti chiavi:

$$Keys = \{C, ED, EB...\}$$

Per cui la relazione è BCNF a meno della dipendenza funzionale  $F \rightarrow B$ .

Si applica nuovamente l'algoritmo:

$$X = F \qquad Y = B \qquad Z = CDE$$

identificando:

$$R3(BF) = \{F \rightarrow B\} \qquad R4(CDEF) = \{C \rightarrow ED, ED \rightarrow F, ED \rightarrow C\}$$

Da R3 si evince immediatamente che la relazione è in BCNF, ma anche R4, facendo il grafo delle relazioni è in BCNF.

Nell'applicazione dell'algoritmo si è persa la dipendenza funzionale  $AB \rightarrow C$ , si capisce dal fatto:

- Non ci sono relazioni con A,B, C insieme
- Non ci sono relazioni da cui è possibile ricongiungere A,B,C

### Esempio

Si faccia ancora riferimento a:

$$Esame(matr, nome, materia, docente, data, voto)$$

$$\left\{ \begin{array}{l} Matr \rightarrow Nome \\ Materia \rightarrow Docente \\ Matr, Materia \rightarrow Data, Voto \end{array} \right. \quad (11)$$

$$Keys = \{Matr - Materia\}$$

La prima dipendenza è tale che la parte sinistra non è una chiave, per cui si applica l'algoritmo visto:

$$X = Matr \qquad Y = X^+ / X = Nome \qquad Z = Materia, Docente, Data, Voto$$

Si realizzano le relazioni:

- $Studente(Mat., Nome)$  da cui  $Matr \rightarrow Nome$

- $Esame'(Matr, Materia, Docente, Data, Voto)$  da cui:

$$\begin{cases} Materia \rightarrow Docente \\ Matr, Materia \rightarrow Data, Voto \end{cases} \quad (12)$$

L'attributo nome, non implica alcun attributo, e questo si può vedere dallo schema delle relazioni, per cui non è necessario fare la proiezione di alcun attributo e quindi inserire una nuova dipendenza funzionale. Questa situazione si verifica nel seguente caso:

Materia non è una chiave candidata per cui è ancora necessario utilizzare l'algoritmo:

$$X = Materia \quad Y = X^+ / X = Docente \quad Z = Data, Voto$$

Da cui si ottengono le relazioni:

- $Insegnamento(Materia, Docente)$  da cui:  $Materia \rightarrow Docente$
- $Esame(Matr, Materia, Data, Voto)$  da cui:  $Matr, Materia \rightarrow Data, Voto$

Le relazioni ottenute sono tutte BCNF, inoltre ogni schema relazionale presenta una dipendenza funzionale iniziale, e tutte le dipendenze funzionali, per cui vengono conservate le informazioni e non perse le dipendenze funzionali

### Esempio

Orario(Aula, Ora, Materia, Docente)

Vale l'insieme di dipendenze funzionali:

$$\begin{cases} Materia \rightarrow Aula, Docente \\ Aula, Ora \rightarrow Materia \end{cases} \quad (13)$$

Dalla quale si ottiene il grafo delle relazioni:

E si ricava l'insieme delle chiavi:

$$Keys = \{Aula - Ora, Ora - Materia\}$$

Materia non è una chiave ed è presente nella parte sinistra di una dipendenza funzionale, si applica l'algoritmo:

$$X = Materia \quad Y = Docente, Aula \quad Z = Ora$$

Si identificano le relazioni:

$R1(Materia, Docente, Aula)$  da cui:  $Materia \rightarrow Docente, Aula$

$R2(Materia, Ora)$  questa relazione non identifica, all'interno del grafico delle relazione, alcuna dipendenza funzionale, per cui si ha l'insieme vuoto.

Sia  $R1$  che  $R2$  sono in BCNF, ma è stata persa la dipendenza funzionale  $Aula, Ora \rightarrow Materia$ , vengono ammesse diverse lezioni nella stessa aula e alla stessa ora.

Quando ci sono situazioni di questo tipo, piuttosto che accettare di perdere dipendenze funzionali, si decide di ammettere ridondanza. Questa forma normale è detta TERZA FORMA NORMALE(3NF).

### DEFINIZIONE

Data una  $\langle R(A_1, \dots, A_n), F \rangle$ ,  $R$  è in 3NF se  $\forall X \rightarrow Y \in F$  in forma canonica vale almeno una delle due:

1.  $X$  è una chiave
2.  $Y$  appartiene a una chiave

Se  $R$  è in BCNF allora è in 3NF. Inoltre esiste sempre una scomposizione in 3NF senza perdita di informazioni e senza perdita di dipendenze funzionali.

Si supponga di avere  $R(ABCDEF)$

SI ha una copertura minimale, da cui si ottengono le chiavi:

$$Keys = \{F, ED, CE, EA\}$$

Dalle dipendenze funzionali si nota che la 1°, 4°, 5° e 6° non sono BCNF:

$$\left\{ \begin{array}{l} AB \rightarrow C \\ CE \rightarrow F \\ F \rightarrow DE \\ C \rightarrow D \\ D \rightarrow A \\ E \rightarrow B \end{array} \right. \quad (14)$$

Si verifica se la relazione è già in 3NF:

- $AB \rightarrow C$

Anche se la parte sinistra non appartiene ad alcuna chiave, la parte sinistra ad EC, per cui questa dipendenza funzionale non viola la 3NF.

- $CE \rightarrow F$

La parte sinistra è una chiave

- $F \rightarrow DE$

La parte sinistra è una chiave

- $C \rightarrow D$

La parte sinistra non è una chiave, ma la parte destra appartiene alla chiave ED

- $D \rightarrow A$

La parte sinistra non è una chiave, ma la parte destra appartiene alla chiave AE

- $E \rightarrow B$

Questa dipendenza funzionale non è in 3NF

Per effettuare la Scomposizione in 3NF si prendono tutte le dipendenze funzionali e si crea una relazione per ogni dipendenza funzionale.

Tale relazione sarà caratterizzata da tutte le dipendenze funzionali che riguardano gli attributi della relazione:

- $R1(ABC) \quad \{AB \rightarrow C, C \rightarrow A\}$

- $R2(CEF) \quad \{CE \rightarrow F, F \rightarrow EC\}$

$F \rightarrow EC$  deriva da  $F \rightarrow E$ , ma Se coincidono in E coincidono anche in B, così come se coincidono in F coincidono anche in D che coincide in A, e AB coincide in C.

- $R3(DEF) \quad \{F \rightarrow DE, DE \rightarrow F\}$

$DE \rightarrow F$ , deriva dal fatto che, se due tuple coincidono in D ed E lo fanno anche in A e B, per cui coincidono in C e di conseguenza in F.

- $R4(CD) \quad \{C \rightarrow D\}$

- $R5(DA) \quad \{D \rightarrow A\}$

- $R6(EB) \quad \{E \rightarrow B\}$

Per garantire che non si perdano informazioni, è necessario almeno una tra le nuove relazioni, contenga una chiave della relazione iniziale. In questo caso la relazione è verificata, altrimenti

sarebbe stato necessario aggiungere una nuova relazione  $R7$  che presenta come attributi, gli attributi di una chiave e un insieme vuoto di dipendenze funzionali.

Si verifichi se esistono coppie di relazioni che è possibile accorpare salvaguardando la 3NF. Prendendo ad esempio  $R5$  e  $R6$ , si va a creare la relazione:

$$R56(ABDE) \quad \{D \rightarrow A, E \rightarrow B\}$$

che presenta l'insieme degli attributi delle due relazioni e il corrispondente insieme delle dipendenze. L'insieme delle dipendenze funzionali può però essere arricchito, aggiungendo:

$$AB \rightarrow D \quad \text{ottenendo l'insieme:} \quad \{D \rightarrow A, E \rightarrow B, AB \rightarrow D\}$$

è però necessario verificare che la dipendenza sia in 3NF. Si considera il grafo delle dipendenze: Si identificano 2 chiavi:

$$Keys = \{EA, ED\}$$

La dipendenza  $E \rightarrow B$ , ha la parte sinistra che non è una chiave, e la parte destra che non appartiene ad alcuna chiave, per cui la relazione  $R56$  non è in 3NF.

In generale, per ridurre il numero di relazioni, conviene accorpare eventuali cicli. Ad esempio si considera il ciclo  $A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$ . Questo ciclo corrisponde ad accorpare  $R1$ ,  $R4$  e  $R5$ :

$$R145(ABCD) \quad \{AB \rightarrow C, C \rightarrow D, D \rightarrow A\}$$

nel grafo delle dipendenze:

si determinano le chiavi:

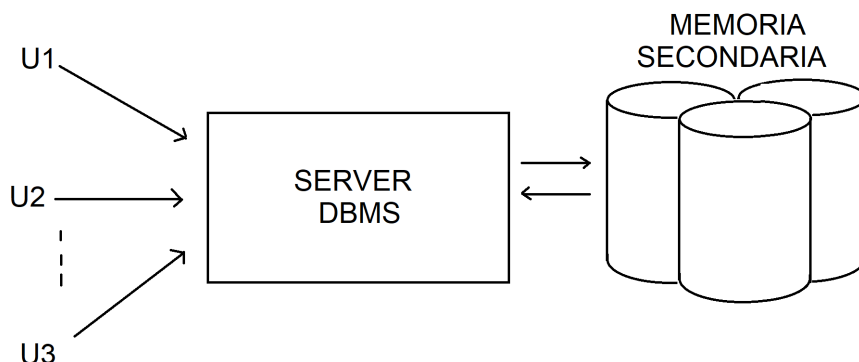
$$Keys = \{AB, CB, DB\}$$

Tutte e 3 le chiavi rispettano le regole affinché  $R145$  sia una relazione in 3NF.



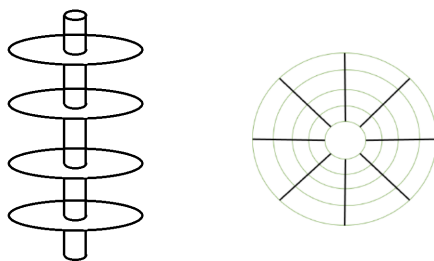
## 8 Implementazione in memoria delle Basi di Dati

I DBMS hanno un architettura di tipo Client-Server, si ha un DBMS Server, a cui arrivano richieste da diversi utenti (che vogliono fare operazioni sul DBMS (insert, delete ecc...), questi interfacciamenti in ingresso sono identificati da Statement in SQL. Il DBMS si appoggia ad una collezione di dati, questi seguono una rappresentazione proprietaria, ovvero ogni DBMS può decidere la codifica dei dati che preferisce (ad esempio la codifica binaria). È importante che l'accesso di più utenti al DBMS venga gestito correttamente senza però esagerare nelle limitazioni di accesso.



La gestione dei dati nelle tabelle avvengono attraverso memoria secondaria, questo condizionerà anche le metodologie di gestione dei dati (diverse dagli algoritmi sui grafi ecc...).

Un Hard Disk è formato da un insieme di dischi che girano intorno l'asse di rotazione, e da una testina, che si può muovere verticalmente, raggiungendo tutti i dischi:

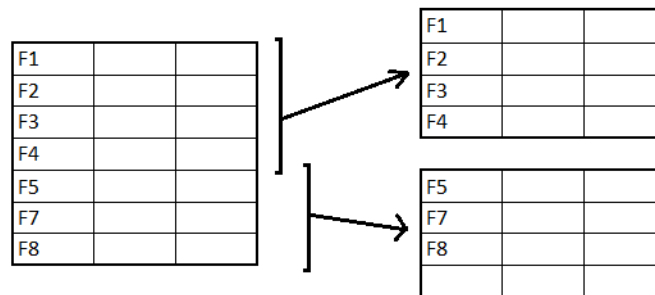


Gli hard disk utilizzano un sistema di coordinate basato su tracce e microsettori. Ogni qual volta è necessario prelevare un dato, è necessario sapere il disco, numero di traccia e settore. Una volta che la testina raggiunge la posizione corretta, deve attendere che il disco specifico ruoti in modo che il dato da prelevare si vada a posizionare proprio sotto la testina.

Questo processo rispetto alla RAM è molto più complesso, quando la testina si muove, posizionandosi su un microsettore, non viene prelevata soltanto una "word" (che in un sistema di calcolo corrisponde alla quantità di BIT processabili nell'unità di clock) a cui si è interessati, ma un intero microsettore (che equivale ad esempio ad 1kB), prelevare più dati comporta ricercare poi il dato

nella porzione prelevata.

Solitamente i dati di una tabella vengono rappresentati in forma paginata, la paginazione si riferisce al fatto, che data una tabella:



questa non è trattata come un file ma come un insieme di pagine di dati, ad esempio le prime 4 tuple sono contenute in una pagina dati così come le secondo 4 tuple, e così via...

Ricercare un particolare dato, in memoria centrale sarebbe lineare (in questo caso costo 7), in memoria secondaria dipende da:

$$2 + x$$

dove 2 è il numero di pagine e  $x$  è il costo di scansione delle tuple di una pagine. Questo è trascurabile in quanto il tempo di accesso a ciascuna pagina è così elevato rispetto alla ricerca che non viene considerato.

Il modo più semplice per inserire le tuple di una tabella all'interno di pagine è il metodo SERIALE. Le tuple sono memorizzate in cascata man mano che vengono fornite, terminato il riempimento di una pagina, viene creata la successiva:



Una struttura di questo tipo, ha un costo:

	Inserimento	Ricerca
Seriale	Theta(1)	Theta(N)

L'inserimento, ha costo  $\theta(1)$ , comporta leggere dalla memoria secondaria la pagina puntata con lo spazio libero, portarla in memoria principale, scrivere la nuova tupla dentro e riscriverla sull'hardisk. In realtà quando viene inserita una tupla bisogna verificare che sia soddisfatto il vincolo di chiave e chiave esterna, per cui il costo è lineare rispetto al numero di pagine (si trascura però questo fattore, in questo momento).

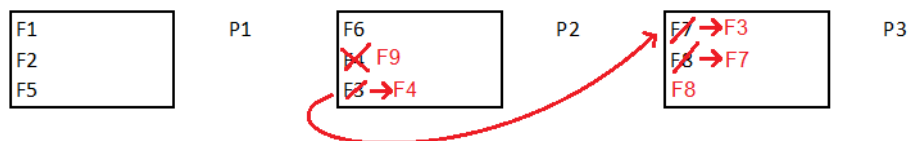
La ricerca è lineare rispetto al numero di pagine, il costo reale sarebbe:

$$N + N * t$$

N pagine vengono portate in memoria centrale e per ognuna è utilizzato un tempo t per scandirle. Inizialmente le memorie secondarie erano i nastri, per cui strutturalmente la memoria era seriale, per cui era naturale utilizzare questa rappresentazione.

Un'altra alternativa può essere rappresentale le tuple in maniera sequenziale, ovvero ordinando in base ad un certo attributo:

### Inserimento F9:



è necessario scandire preliminarmente le pagine, trovata la posizione in cui inserire una nuova tupla, vengono scalate le altre.

In generale i costi per il caso sequenziale, sono i seguenti:

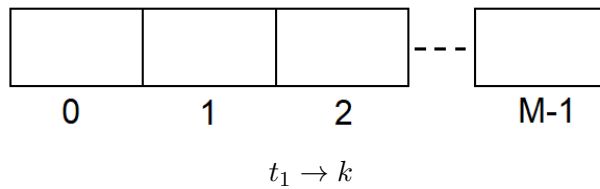
	Inserimento	Ricerca
Sequenziale	Theta(N)	Theta(N)

La dipendenza dal numero di pagine N, non è efficiente, si passa ad altre strutture.

Una su tutte le TABELLE HASH, da rivisitare, in quanto i costi di accesso non sono legati ai numeri di tuple ma alle pagine.

Nella tabella Hash si hanno un insieme di pagine indicizzate:

L'analisi che verrà fatta, si riferirà alla ricerca di una tupla. Ogni volta che arriva una tupla si trasforma in una chiave k:



dove  $k$  è la chiave di accesso che da informazioni sulla pagina in cui è contenuta la tupla.

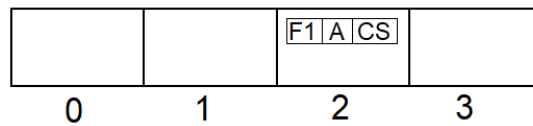
es

Si supponga di avere  $M=4$  all'arrivo di  $t_1$ , questa ha una chiave:

$$k_1 = 6$$

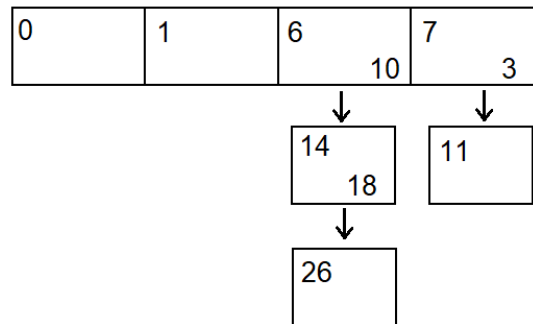
La tupla  $t_1$  viene inserita nella pagina:

$$H(4) = 6 \bmod 4 = 2$$



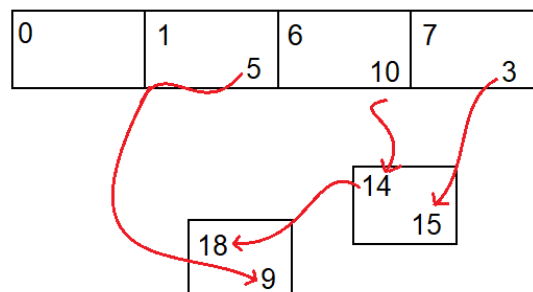
In questo caso tutte le pagine hanno spazio disponibile per l'inserimento, esistono però politiche per la gestione dei trabocchi (ovvero quando una chiave riconduce ad una pagina satura).

Nel momento in cui una pagina risulta satura, ad essa è assegnato un puntatore in cui vengono messi i trabocchi della pagina:

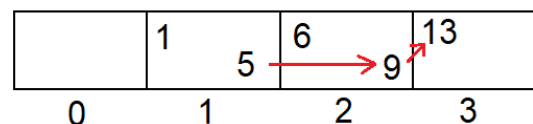


Il tempo di accesso per la ricerca/inserimento di una tupla, nel caso migliore è costante, se la pagina non ha trabocchi, altrimenti sarà lineare rispetto la dimensione della lista di trabocchi(per semplicità nelle pagine si indicano le chiavi e non le tuple).

Un'ulteriore soluzione, per evitare (in seguito a numerose cancellazioni), di avere FRAMMENTAZIONE, è quella di non creare una lista di trabocchi per ogni pagina, ma avere delle pagine comuni, dove le tuple sono caratterizzate da puntatori personali per identificare la pagina originale di provenienza:



Un' altra possibilità è quella di non creare le pagine al di fuori delle M pagine, nel momento in cui una pagina risulta completa, si va ad inserire la tupla nella pagina meno scarica con puntatore alla pagina originaria:



Utilizzando la stessa area dati, si aumenta il fattore di caricamento delle pagine; inoltre, mentre il DBMS fa in modo che, nel momento della creazione delle M pagine su cui viene fatto l'hashing, queste siano facilmente raggiungibili tra loro, quando vengono create pagine per trabocco queste vengono posizionate in memoria dove si ha spazio, di conseguenza potrebbero finire molto lontano dalle altre

## 8.1 Hashing Lineare a Indirizzamento Aperto

Senza memorizzare fisicamente il puntatore delle tuple di trabocco, viene utilizzato un algoritmo che aumenta il valore di  $k$  da posizionare con un valore  $i$ :

$$H(k) = k \bmod M \quad k = k + i$$

È necessario memorizzare i valori ' $i$ ' in modo da poter ricomporre le liste di trabocco in maniera completa.

Questo sistema è più leggero del precedente, in quanto non dovendo memorizzare i puntatori, ogni pagina consente di memorizzare più tuple.

Ad ogni modo comporta il fenomeno di AGGLOMERAZIONE PRIMARIA, date 4 pagine, inizialmente ognuna ha equivalente probabilità di accogliere una tupla di trabocco:

25%	25%	25%	25%
-----	-----	-----	-----

Supponendo che la pagina 1 si riempia le percentuali si ridistribuiscono nel seguente modo:

25%	0%	50%	25%
-----	----	-----	-----

e nel caso in cui anche la pagina 2 si riempirà, si otterrà la situazione:

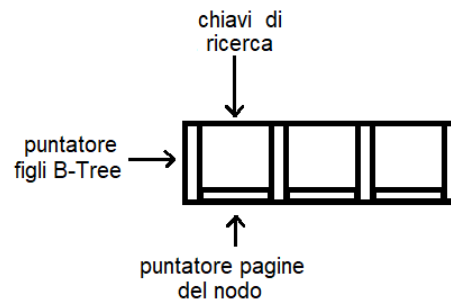
In sostanza, con l'aumentare dei trabocchi, questa tecnica tende a riempire delle pagine, che risultano destinazione più probabili, mentre l'hashing funziona correttamente quando si ha equiprobabilità. Col tempo questo sistema tende di generare fenomeni di TRABOCCHI a CATENA. Per evitare questo problema, spesso il valore ' $i$ ' viene selezionato in maniera casuale.

Si noti che questa tecnica, come le precedenti viste, sono tecniche di HASHING STATICO, ovvero che richiedono di conoscere a priori il numero di pagine.

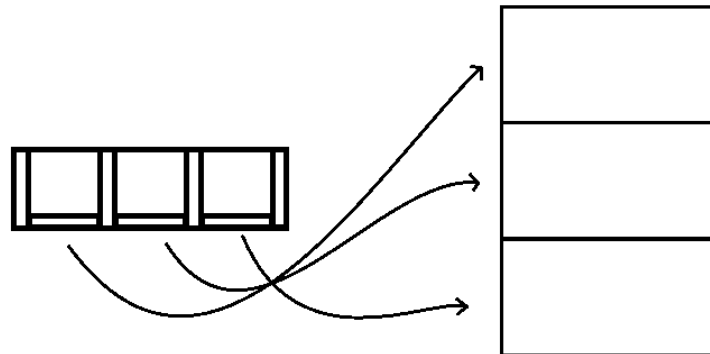
## 8.2 B-Tree

Un B-Tree è una struttura che si basa sulla logica dell'albero.

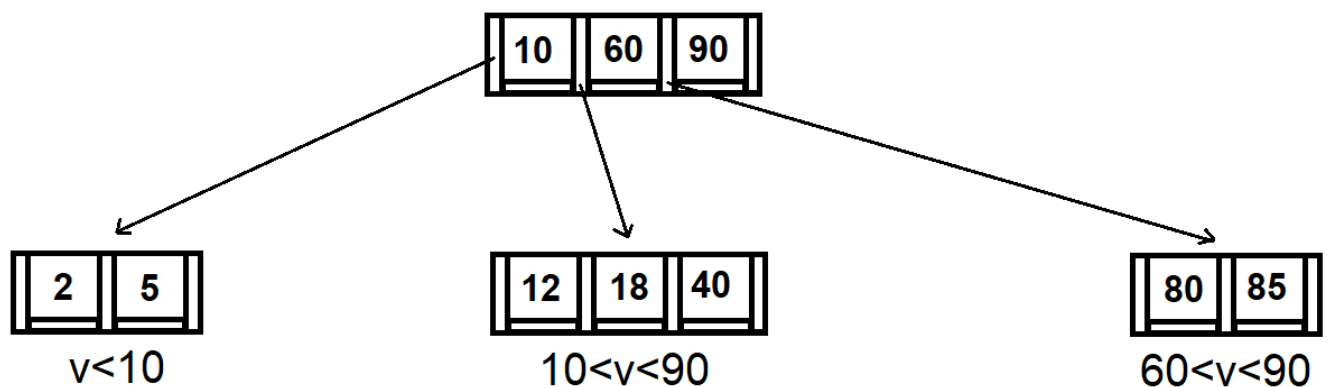
Ogni nodo di un B-Tree ha forma:



ogni nodo è un puntatore ad una pagina, contenente più tuple ORDINATE:



I nodi FOGLIA non presentano a sinistra e destra puntatori a chiavi. Ogni nodo FIGLIO deve mantenere l'ordinamento delle chiavi dell'albero di ricerca, ovvero contenere valori compresi tra il sinistra e destra del nodo padre:



Date  $x$  chiavi si hanno al più  $x+1$  figli.

Il **GRADO** dell'albero rappresenta il numero di chiavi che ciascun nodo può contenere, ovvero il numero di figli che può avere.

In un sistema DBMS quando si vuole creare un B-Tree per velocizzare le ricerche sul nome di un

fornitore, si può utilizzare lo statement:

```
CREATE INDEX  
ON Fornitore (nome)  
...
```

viene preso l'attributo nome, viene visto il tipo di dato (tipologia e di conseguenza dimensione) e si crea un B-Tree con grado corrispondente.

La ricerca avviene accedendo ad ogni passo, al nodo del livello corrente del B-Tree. Se in una foglia non viene trovata la chiave cercata, si conclude che la chiave non è presente nel database.

La struttura del B-Tree consente con costi ridotti di vedere la presenza o meno di una tupla nel database, in modo da dare errore nel caso in cui esiste già, ma anche per verificarne l'esistenza quando una chiave viene usata come chiave esterna.

Il costo di accesso ad un B-Tree è pari all'altezza dell'albero, che se bilanciato, ha costo:

$$\log_{\text{grado}} n$$

La dipendenza del logaritmo dal grado, consente di indicizzare grandi quantità di tuple in alberi con altezza piccola, si pensi che 1000000 di tuple in un B-Tree di grado 100 sono indicizzabili su 3 livelli.

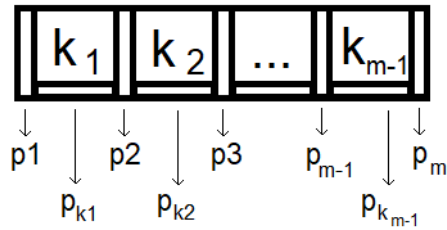
Inoltre avendo un grado molto ampio, i B-Tree hanno una base molto larga e la gran parte dei nodi si trova all'ultimo livello, di conseguenza i nodi contenuti dal penultimo livello in SU occupano poco e possono essere tenuti in memoria principale.

Bisogna stare attenti, che i nodi del B-Tree siano abbastanza pieni, in modo da evitare che le pagine siano SPARSE.



### 8.2.1 Caratteristiche B-Tree di grado m

1. Ogni nodo ha la struttura:



2. Ogni nodo ha AL PIÙ m figli
3.  $k_1, \dots, k_{m-1}$  sono ordinate
4. Il sotto albero puntato da  $p_i$  contiene chiavi maggiori di  $k_{i-1}$  (se  $i > 1$ ) e minori di  $k_i$  (se  $i = m$ )
5. Se un nodo ha x chiavi e non è foglia, allora ha x+1 figli  
Se un nodo ha x figli, allora ha x-1 chiavi
6. Tutte le foglie sono alla stessa altezza, per cui preso un nodo, la differenza tra l'altezza dell'albero si SX e DX è pari a 0. Questa è una forte proprietà di BILANCIAMENTO.
7. Ogni nodo non radice possiede almeno  $m/2 - 1$  chiavi, questo garantisce che l'albero sia sparso.

### 8.2.2 Inserimento nel B-Tree

Si consideri di partire da un B-Tree di grado 4 vuoto.

- Inserimento di 10



- Inserimento di 20



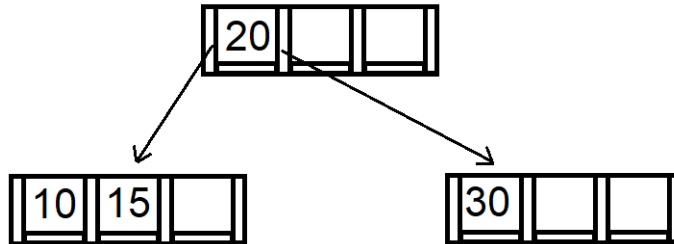
- Inserimento di 15



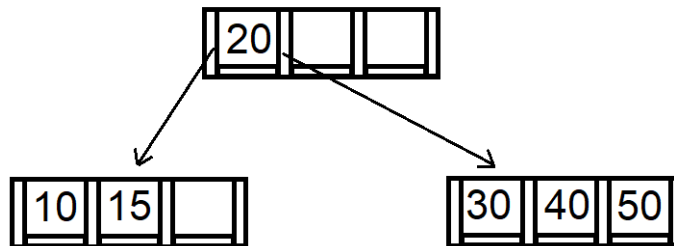
- Inserimento di 30 Non è presente spazio per l'inserimento, si considerano tutti i valori:

10,15,20,30

Si identifica il valore medio (20) e si fa salire nel B-Tree:



- Inserimento di 40, 50

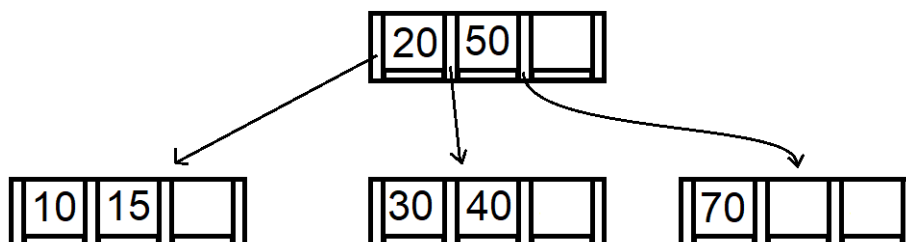


- Inserimento di 70

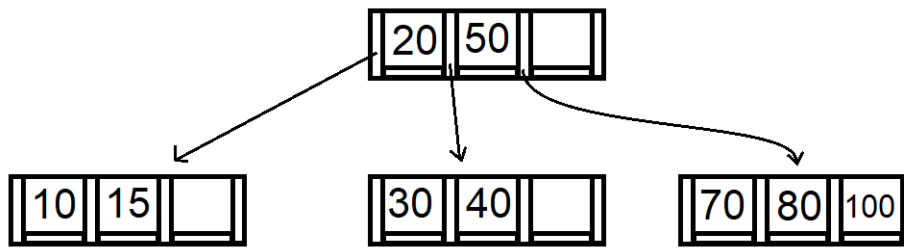
Il nodo foglia "esplode", si prendono i valori:

30,40,50,70

Il nodo 50 sale i valori a destra del 50 andranno nel suo figlio destro, mentre i valori a sinistra nel figlio sinistro:



- Inserimento 80,100

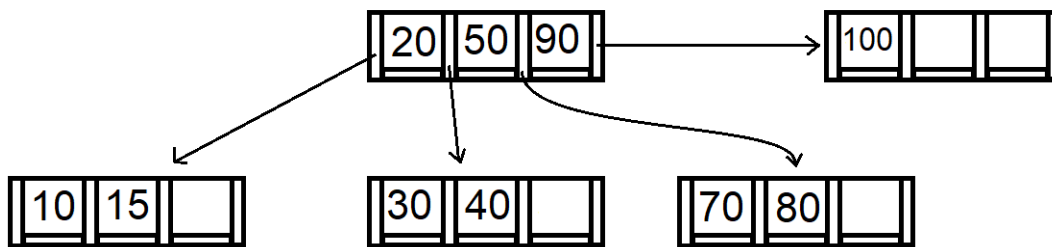


- Inserimento 90

Il nodo foglia esplode nuovamente:

70, 80, 90, 100

90 sale e si ottiene il B-Tree finale:

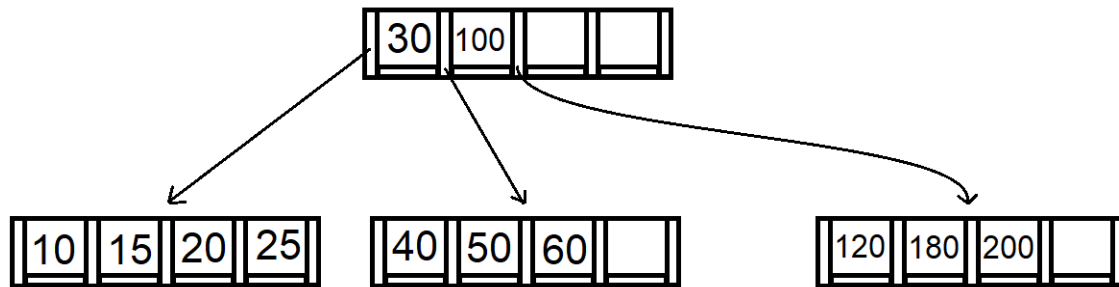


In **GENERALE** l'albero cresce dal basso, per ogni inserimento è necessario fare:

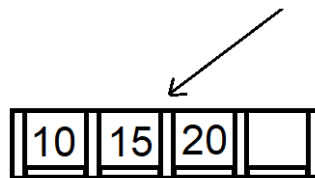
- $h$  letture
- $2h+1$  scritture

### 8.2.3 Eliminazione di un nodo

Si consideri il B-Tree:



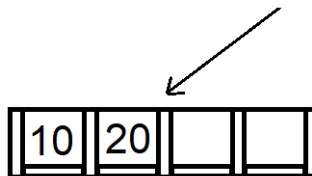
- Rimozione 25



Il riempimento minimo è garantito , dato  $m$ , devono essere presenti almeno:

$$\frac{m}{2} - 1 \text{ chiavi} \quad \text{OK!}$$

- Rimozione 15



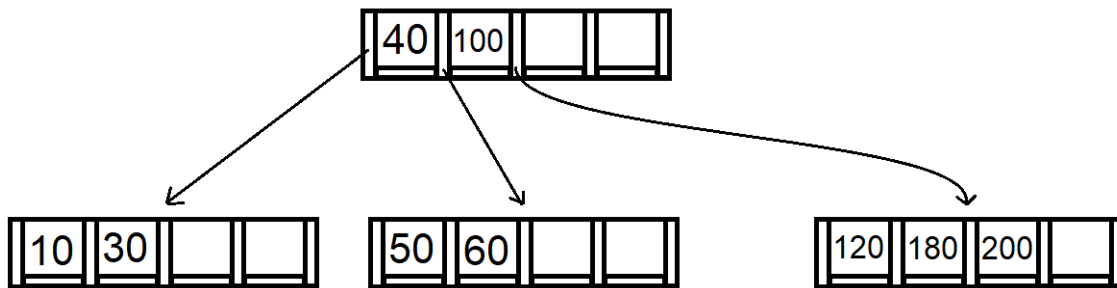
Il riempimento minimo è garantito.

- Rimozione 20

Il riempimento minimo non è più garantito, il nodo "implode". Si prendono i valori del nodo padre e del figlio DX del padre:

10, 30, 40, 50, 60

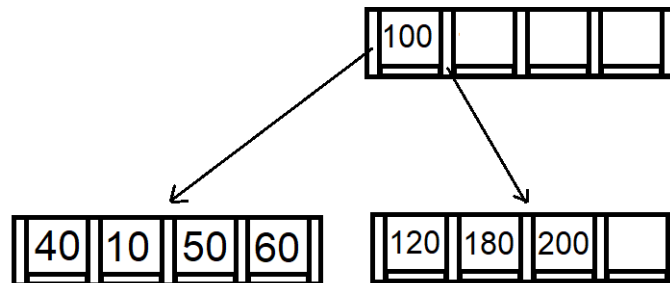
Il nodo 40 sale e si ottiene:



Eliminando ora il nodo 30, si creerebbe una situazione particolare, in quanto se il 50 salisse e il 40 assumesse il posto lasciato libero da 30, ci sarebbe comunque un nodo a riempimento NON minimo. Non basta quindi la solita rotazione, ma è possibile accorpare:

10,40,50,60

in un nuovo nodo, si effettua quindi una cancellazione al livello superiore:



A questo punto si vede se il nodo superiore verifica la condizione di riempimento minimo, in maniera ricorsiva. In questo caso è una radice di conseguenza SI.

Man mano che si va avanti, l'albero scende di altezza.

### 8.2.4 Tuple indicizzabili e Altezza

Si supponga di avere un B-Tree di grado  $m$  ed altezza  $h$ , e che abbia riempimento massimo, ovvero, tutti i nodi hanno  $m$  figli, per cui si ha un numero di pagine:

$$P_{MAX} = 1 + m + m^2 + \dots + m^{h-1}$$

che è una progressione geometrica:

$$= \frac{m^h - 1}{m - 1}$$

Il numero di tuple indicizzabili è pari a:

$$N_{MAX} = (m - 1)P_{MAX} = m^h - 1$$

Il numero di tuple effettive presenti sarà:

$$\begin{aligned} N &\leq m^h - 1 \\ &\downarrow \\ h &\geq \log_m(N + 1) \end{aligned}$$

Se il riempimento è minimo:

$$\begin{aligned} P_{MIN} &= 1 + 2 + \left\lceil \frac{m}{2} \right\rceil + 2 \left\lceil \frac{m}{2} \right\rceil^2 + 2 \left\lceil \frac{m}{2} \right\rceil^{h-2} = \\ &= 1 + 2 \left\{ \left\lceil \frac{m}{2} \right\rceil + 2 \left\lceil \frac{m}{2} \right\rceil^2 + 2 \left\lceil \frac{m}{2} \right\rceil^{h-2} \right\} = \\ &= 1 + 2 \left( \frac{\left\lceil \frac{m}{2} \right\rceil^{h-1} - 1}{\left\lceil \frac{m}{2} \right\rceil - 1} \right) \end{aligned}$$

Il numero di tuple corrispondente sarà:

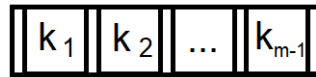
$$N_{min} = 1 + (P_{min} - 1) \left( \left\lceil \frac{m}{2} \right\rceil - 1 \right) = 2 \left\lceil \frac{m}{2} \right\rceil^{h-1} - 1$$

Il numero di tuple effettivo sarà:

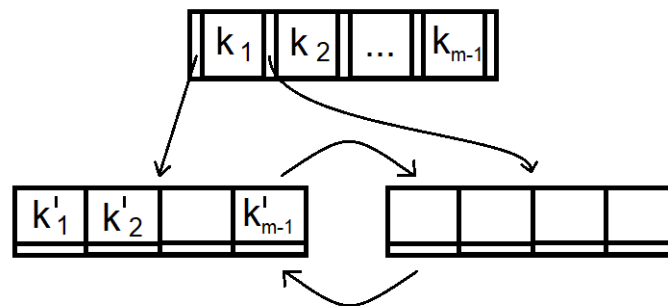
$$\begin{aligned} N &\geq 2 \left\lceil \frac{m}{2} \right\rceil^{h-1} - 1 \\ &\downarrow \\ h &\leq 1 + \log_{\left\lceil \frac{m}{2} \right\rceil} \left( \frac{N + 1}{2} \right) \end{aligned}$$

### 8.3 $B^+$ Tree

Nel  $B^+$ -Tree vale lo stesso criterio di ordinamento di un B-Tree, ma i nodi intermedi hanno una struttura in cui si hanno una serie di chiavi e una serie di puntatori ai figli.

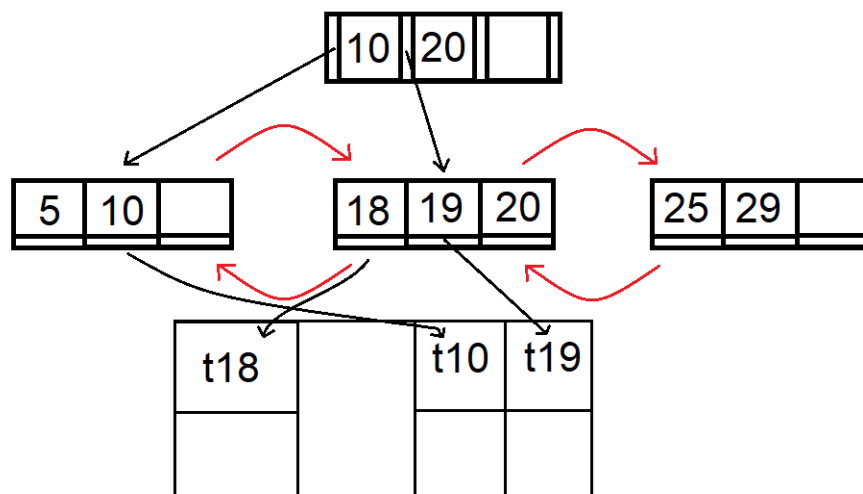


Mentre nel B-Tree le chiavi presenti ad un livello intermedio possiedono già un puntatore ad una tupla nella loro posizione, nel  $B^+$ -Tree si trovano comunque ripetute ad un livello foglia. Inoltre i nodi a livello foglia hanno la caratteristica di avere dei puntatori che generano una lista concatenata, questo vuol dire che nell'ultimo livello, sono riportate le chiavi indicizzate ordinate in senso lessicografico rispetto le chiavi di ricerca.



#### Esempio

Si supponga di avere un  $B^+$ -Tree di ordine 4:



è presente una tabella di pagine, contenenti le tuple, puntate dalle foglie dell'albero.

Tutte le operazioni di inserimento/rimozione che influenzano sul bilanciamento si fanno con maggior facilità rispetto al B-Tree.

I nodi all'ultimo livello possiedono un puntatore in avanti/indietro tra le foglie che si trovano allo stesso livello, questo rende il  $B^+$ -Tree molto comodo per effettuare ricerche per rang.

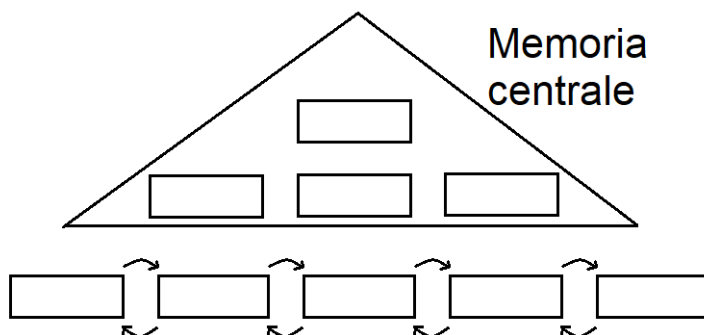
Si supponga di voler trovare i valori compresi tra 10 e 20, trovato il valore 10 e la sua posizione nella foglia, basta scandire le foglie fino a quando non si trova il valore 20.

A livello di accessi, si accede alla radice e a 2 nodi che porta all'accesso alle  $n$  pagine delle  $n$  tuple presenti nell'intervallo.

Anche nel B-Tree era possibile realizzare le query basate su range, ma non essendo le tuple tutte a livello foglia, era necessario utilizzare degli iteratori su albero.

Un nodo intermedio nel  $B^+$ -Tree indicizza più chiavi rispetto al B-Tree in quanto si risparmia lo spazio dei puntatori (ad esempio il puntatore di 10), a parità di contesto un nodo intermedio del  $B^+$ -Tree ha un grado maggiore, questo crea il vantaggio di avere un ALBERO PIÙ BASSO.

Più è bassa l'altezza ed  $M$  è grande, più si ci avvicina alla configurazione:



Dove tutti i nodi stanno in memoria centrale a meno di quelli dell'ultimo livello che sono in memoria secondaria. Questa situazione è più raggiungibile in quanto lo scarto tra numero di nodi tra ultimo livello e superiori è ancor maggiore (essendo  $M$  più grande).

Questa configurazione rende trascurabile il costo di discesa e salita sull'albero in quanto avvengono in memoria centrale.

Il fatto che ogni chiave di un livello intermedio è ripetuta due volte non pesa troppo sulla struttura, in quanto come già detto, la maggior parte delle chiavi è contenuta nell'ultimo livello dell'albero per cui sono poche le chiavi ripetute.

L'indicizzazione tra dati dell'albero e tuple è staccato, l'albero può essere ordinato ad es. per PIVA, mentre le tuple sul nome del cliente. Si può scegliere di ordinare indici e tuple secondo lo stesso criterio, in questo modo nel momento in cui si effettua una ricerca per range, basterà accedere alla pagina iniziale dell'intervallo da analizzare e poi scandire le pagine dati, senza dover ogni volta raggiungere la pagina partendo dall'indice.



La configurazione per cui indici e tuple sono ordinati in base alla stessa chiave candidata ottimizza gli accessi e prende il nome di  $B^+$ -Tree con **Indice Clustered**

Un'altra configurazione è quella con **Indice Sparso**, in cui non sono memorizzate tutte le chiavi delle tuple, ma solo alcune (ad esempio 1/4 delle chiavi), e più tuple sono memorizzate nella stessa pagina. Nel momento in cui è necessario trovare una tupla, si usa un indice per trovare una posizione ragionevole di dove si possa trovare nelle pagine dati, scandendo da quella posizione.

Quando si crea un indice su un attributo o più attributi non chiave rispetto ai quali non si ha una memorizzazione ordinata delle tuple, si ha un livello intermedio tra foglie del B-Tree e tuple. Il livello è basato su tabella, dove ogni campo della tabella contiene un valore dell'attributo di indicizzazione e i puntatori alle tuple dell'area dati che presentano tale valore.

## 9 Modello di Dati Semi-Strutturato

Un modello relazionale è un:

- modello FORTEMENTE STRUTTURATO, ovvero qualsiasi informazione si voglia rappresentare in un DBMS questa deve essere codificata in una struttura rigida ben precisa.
- La struttura dati tipicamente non varia, al più si adattano i dati allo schema
- Lo schema e i dati sono ben distinti tra loro (Si ha lo schema relazionale e l'istanza di uno schema)
- Lo schema è più piccolo dei dati

Non tutti i dati possono essere però rappresentati con modelli relazionali, in particolare informazioni che provengono ad esempio da diverse basi di dati.

Nel modello semi strutturato:

- La struttura è lasca, non tutti i dati che rappresentano informazioni riguardanti lo stesso concetto, hanno la stessa forma
- La struttura può variare nel tempo
- Lo schema può essere codificato nei dati
- Lo schema può avere dimensioni paragonabili ai dati

Non si chiama modello destrutturato in quanto è presente una struttura ma questa non è ben definita.

## 9.1 XML

Il linguaggio standard per il modello semistutturato è il linguaggio XML ( eXtensible Markup Language ), ovvero linguaggio di limitatore estendibile. Il linguaggio ha una logica comune all' HTML che si occupa di linguaggio di limitatori per ipertesti, ovvero entrambi usano dei limitatori. Un file HTML ha una struttura del tipo:

```
<HTML>
  <HEAD>

  <\HEAD>

  <BODY>

  <\BODY>
<\HTML>
```

Tutto è racchiuso tra dei delimitatori INNESTATI, inoltre la visualizzazione della pagina non mostra nulla di quello che è presente nell'Head (sono detti metadati) ma solamente quello che è presente nel Body.

Allo stesso modo in XML sono utilizzati dei delimitatori per definire la struttura del dato che si sta rappresentando.

Un esempio di un documento XML molto semplice è il seguente:

```
<biblioteca>
  <libro>
    <titolo> La Divina Commedia <\titolo>
    <autore> Dante <\autore>
  <\libro>

  <libro ID="2">
    <ISBN> 348-11- CB <\ISBN>
    <autori>
      <autore> Fontero <\autore>
      <autore> Lucentini <\autore>
    <\autori>
```

```

    <titolo> Un caso difficile <\titolo>
    <casaEditrice> Sinterio <\casaEditrice>
  <\libro>
<\biblioteca>

```

Una biblioteca è una lista di libri, ogni libro però può essere caratterizzato da diversi delimitatori o TAG. Tutto ciò racchiuso in una TAG è detto Elemento. Ogni oggetto può essere anche caratterizzato da degli attributi come nel caso del 2° libro l'ID. Tutti i tag potrebbero essere messi come attributi ma poi il linguaggio perderebbe la sua caratteristica.

Il documento scritto ha una codifica testuale e non binaria come nei Database.

È chiaro che in HTML i delimitatori sono di sintassi, mentre in XML li crea l'utente.

Quando si pubblica un documento XML, lo si corredda con una GUIDA DEI DATI. Un modo per definirla, è usare il linguaggio DTD(Data Typy Definition) che definisce come sono composti i dati rappresentati all'interno del file XML associato, per far questo da una definizione eventualmente lasca del contenuto di ciascun elemento.

Facendo riferimento al file XML precedentemente scritto, un esempio di DTD è il seguente:

```

<!ELEMENT biblioteca(libro*)>

<!ELEMENT libro(ISBN?, titolo , autori|autore , casaEd?)>

<!ELEMENT autori(autore+)>

<!ELEMENT ISBN #PCDATA>

<!ATTLIST libro identificatore ID
              prezzo CDATA
              curatore IDREF>

```

Questo indica le caratteristiche comuni / possibili dei libri, ad esempio:

- "libro\*" indica che possono esserci da 0 a n libri
- "ISBN?" indica che un libro può avere come non questo limitatore
- "autori—autore" indica che un libro può avere uno o più autori

- "PCDATA" sta per Parsel Character Data, ovvero dati fatti da caratteri sottoposti a pharsing, ovvero gli escape character(caratteri speciali) non vengono visualizzati
- "ATTLIST" fa riferimento alla descrizione degli attributi, che possono essere di piu tipi:
  - ID, identificati per l'appunto da un ID
  - CDATA, di tipo testuale
  - IDREF o IDREFS deve riportare un ID presente all'interno del documento (rappresenta una sorta di chiave esterna)

La scrittura del tipo:

`<!ELEMENT a( b,(c—d)*,(d—e)?,f+)>`

indica che 'a' puo avere forma:

`<a>`

`<b> ... <\b>`

`<c> ... <\c>`

`<c> ... <\c>`

`<d> ... <\d>`

`<e> ... <\e>`

`<f> ... <\f>`

`<\a>`

## Esempio

```
<biblioteca>
  <libro ID="L1">
    <titolo> TL1 </titolo>
    <autore> AL1 </autore>
  </libro>
  < libro ID="L2" casaEd="CE1">
    <titolo> TL2 </titolo>
    <autori>
      <autore> AL2 </autore>
      <autore> AL3 </autore>
    </autori>
    <annoEd> 2020 </annoEd>
  </libro>
  <casaEd ID="CE1">
    nomeCasaEd1
  </casaEd>
</biblioteca>
```

Il DTD associato è il seguente:

```
<!ELEMENT biblioteca (libro+, casaEd+)>
<!ELEMENT libro (titolo , autore|autori , annoEd?)>
<!ELEMENT autori (autore+)>
<!ATTLIST libro ID ID
               casaEd IDREF>
<!ELEMENT casaEd #PCDATA>
<!ATTLIST casaEd ID ID>
```

.  
.  
.

### 9.1.1 Elemento Misto

Un elemento del tipo:

```
<a>
  Stringa 1
  <b>      <\b>
  Stringa 2
  <c>      <\c>
<\a>
```

è detto **ELEMENTO MISTO**, in quanto sono presenti sia stringhe che sotto elementi.

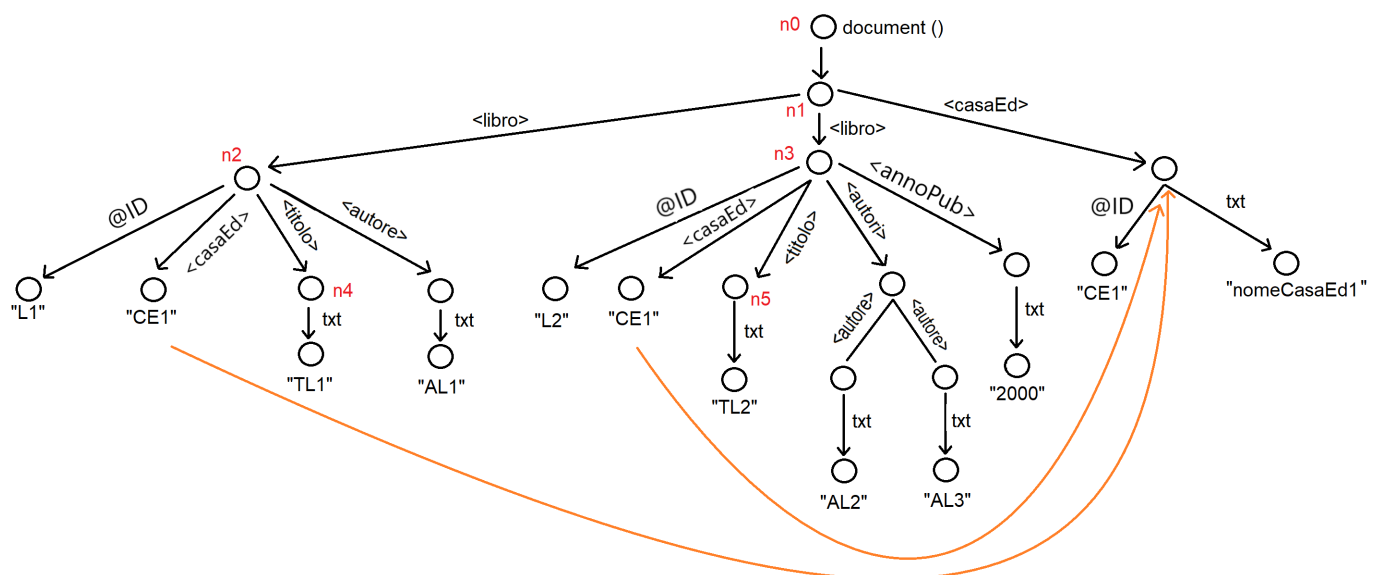
All'interno di un DTD si indica nella forma:

$$<!\text{ELEMENT } a[(b-c-d-PCDATA)^*]>$$

Il DTD non è l'unico modo per descrivere gli elementi di un file XML, esiste anche l'XML Schema, che permette di definire la struttura di un documento in forma molto più precisa.

### 9.1.2 Albero del documento

Ad ogni file XML è possibile associare un albero:



Un documento XML è ben formato quando ha una struttura formata corretta. Ogni nodo, se non foglia, rappresenta un limitatore, mentre ogni nodo foglia il contenuto dello specifico limitatore. Gli attributi di un elemento vengono visti come figli che all'interno hanno solo del testo. Per distinguerli viene usata la chiocciola.

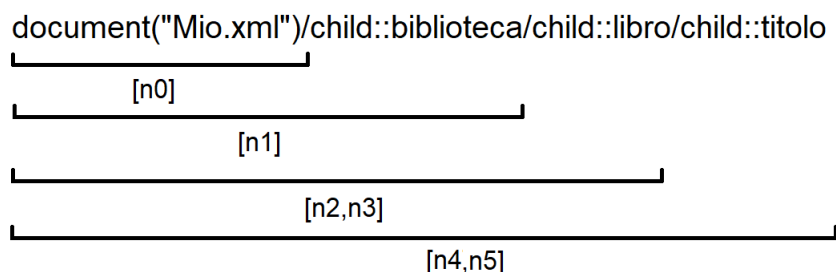
Gli archi arancioni (o tratteggiati) rappresentano dei riferimenti, non considerandoli si viene a creare un grafo connesso aciclico diretto(in quanto si ha distinzione gerarchica tra padre e figlio).

## 9.2 XPath

Ogni qual volta è necessario estrarre qualcosa a cui l'utente è interessato, tramite XPath è possibile specificare il cammino che occorre percorrere dalla radice dell'albero per raggiungere i nodi che hanno le informazioni di interesse.

Il cammino viene rappresentato rappresentando gli assi navigazionali.

Per conoscere i titoli presenti nel documento:



La query restituirà questo output:

```
< titolo>TL1 < / titolo> < titolo> TL2 < / titolo>
```

Un'alternativa è la scrittura:

```
document()/descendant::titolo
```

restituisce tutti i nodi discendenti di document, che all'ultimo passo sono raggiungibile con un nodo titolo, i due risultati coincidono.

Questo tipo di Query trovano maggior senso nel conoscere gli autori, nei due libri, infatti, per raggiungere gli autori servirebbero due query diverse, in quanto l'albero è composto in modo diverso, il problema si risolve con la query:

```
document()/descendant::autore
```

I condandi principali per le query sono:

- child
- descendant
- ancestor, dato un nodo risale l'albero (contrario di descendant)



- parent (contario di child)
- sibling, per muoversi da un nodo ai fratelli

Se invece di:

$$< a > \dots < /a >$$

si vuole soltanto il testo, alla query si aggiunge:

$$\dots/text()$$

naturalmente se il nodo non ha parte testuale, viene restituita una stringa vuota.

Il VALORE TESTUALE di un nodo è la concatenazione di tutte le stringhe che si trovano al suo interno.

Per estrarre gli attributi, si utilizza l'asse di navigazione @ .Ad esempio:

$$\text{document()}//\text{libro}/@ID$$

gli attributi non hanno struttura, per cui restituire un attributo significa restituire il valore della stringa inserita nell'attributo.

Le condizioni, in XPath vengono espresse nelle parentesi quadre, e prendono il nome di **FILTRO DI SELEZIONE**. Un esempio è il seguente:

$$\text{document()}//\text{libro}[\text{./annoPub}/\text{text()}=2000]$$

che a livello descrittivo:

"Partendo da documentano e si prendano i nodi di tipo libro, questi devono essere restituiti se a partire da questo path(.), il libro ha un anno di pubblicazione e questo è 2000"

la query restituirà l'intero libro, è però possibile continuare la query selezionando un particolare limitatore:

$$\text{document()}//\text{libro}[\text{./annoPub}/\text{text()}=2000]/\text{titolo}$$

Quando l'asse che si scrive è figlio, il ./ può non essere inserito, altrimenti è fondamentale.

Il nome del tipo di un elemento può essere sostituito con un nome generico, utilizzando il simbolo di **WILDCARD**:

$$\text{document()}//*[annoPub]/titolo$$

prende tutti i discendenti di documento proseguendo solo su quelli che hanno un figlio "annoPub", non è molto efficiente, in quanto viene considerato ad esempio anche il nodo "casaEd" ma fa risparmiare sulla scrittura della query.

Si supponga di avere un documento rappresentato solo con il DTD:

```
<!ELEMENT collezione(libro+, rivista+)>
<!ELEMENT libro(titolo , autore)>
<!ELEMENT rivista(titolo , articolo+)>
<!ELEMENT articolo(nome, autore , revisori)>
<!ELEMENT revisori(revisore+)>
<!ELEMENT autore(nome)>
<!ELEMENT revisore(nome)>
```

” Nomi delle persone coinvolte nelle riviste”

la scrittura:

```
document()//rivista//nome
```

è errata, in quanto ogni rivista ha nei discendenti il nome dell’articolo , degli autori e dei revisori, è necessario essere più precisi:

```
document()//articolo*/nome
```

Possono essere scritte anche query complesse ed innestabili:

```
document()//a[b/c=”10”][./d=../f]/q
```

dal nodo documento prende tutti i discendenti ”a” a cui applica due filtri:

- il nodo c discendete da b figlio di a deve avere valore pari a 10
- i discendenti di tipo d devono coincidere con i discendenti di tipo f

I filtri restituiscono true se esiste almeno una coppia di valori che li verifica. In particolare, possono esserci più c (come nel caso precedente più anni di pubblicazione):

```
[10 50 32]=? 10
```

restituisce true, ma lo fa anche nel caso in cui ci fosse stato un !=

### 9.3 XQuery

Il linguaggio XQuery è un linguaggio basato su alcune clausole, come SQL, ma mentre SQL deve seguire un ordine preciso (Structured):

```
SELECT
FROM
WHERE
GROUP BY
HAVING
```

Xquery è più libero.

La query più semplice è il:

```
return
```

es.

```
return {document(...)//publisher}
      ↓
      <publisher> ... </publisher>
      .
      .
      <publisher> ... </publisher>
```

Si noti che la return prende come argomento una query di XPath. Si può mettere il risultato all'interno di una coppia di tag:

```
return <risultato> {document(...) //publisher} </risultato>
```

L'espressione da interpretare è inserita tra parentesi graffe, altrimenti XQuery lo valuta a livello testuale e non logico.

Un'altra clausola è il:

```
for
```

consente di scansionare il contenuto di un documento.

es.

```
for $p in document (...) // publisher
return <casaEditrice>
    \ $p/text () $}
</casaEditrice>
```

si chiede di usare una variabile `$p` ed assegnare volta per volta un valore tra i publisher presenti nel documento, il contenuto di `$p` deve ogni volta essere inserito all'interno del tag `casaEditrice`. Si ottiene:

```
<casaEditrice> ... </casaEditrice>
.
<casaEditrice> ... </casaEditrice>
```

Qualora si voglia includere nel tag risultato si modifica la query nel seguente modo:

```
<risultato>
{for $p in document (...) // publisher
return <casaEditrice>
    \ $p/text () $}
</casaEditrice>
}
</risultato>
```

Il linguaggio XQuery è un linguaggio di **SCRIPTING** ovvero può essere integrato all'interno di un documento XML, quando si vuole che una porzione sia interpretata invece che scritta letteralmente (carattere per carattere) si inseriscono le parentesi graffe, per cui non ha senso inserire nella query "return <risultato> ... </risultato>". Tutte le variabili iniziano con il `$`

Un'altra clausola è la **WHERE** che serve ad inserire una condizione:

```
<risultato>
{for $b in document (...) // book
where $b/@year > "2000" $
return <titolo> { $b/title/text () $} </titolo>
}
</risultato>
```

In realtà questa query poteva essere scritta anche senza utilizzare la clausola where ma utilizzando XPath:

```
<risultato>
{for $b in document(...)//book[@year>"2000"]
return <titolo> {$b/title/text()} </titolo>
}
</risultato>
```

```
for $b in document(...)//book
let $a:= $b//author
return <libro>
    <titolo> {$b/title/text()}</titolo>
    <autori> {$a} </autori>
</libro>
```

che restituisce:

```
<libro>
  <titolo> T1 </titolo>
  <autori>
    <author> <last> l1 </last> <first> f1 </first> </author>
    <author> <last> l1 </last> <first> f1 </first> </author>
    <author> <last> l1 </last> <first> f1 </first> </author>
  </autori>
</libro>
```

Se il libro non ha autori la \$a è vuota per cui è presente il tag senza elementi.

Se si volesse ottenere solo il cognome degli autori, è necessario lavorare su \$a inserendo una nuova query:

```
for $b in document(...)//book
let $a:= $b//author
return <libro>
    <titolo> {$b/title/text()}</titolo>
    <autori>
        {for $x in $a
         return zcognomeAutore> {$x/last/text()} </cognomeAutore>
        }
    </autori>
</libro>
```

↓

```
<libro>
    <titolo> T1 </titolo>
    <autori>
        <cognomeAutore> ... <\cognomeAutore>
        .
        .
        <cognomeAutore> ... <\cognomeAutore>
    </autori>
</libro>
```

Un'altro esempio è il seguente:

```
<risultato>
{for $b in document(...) // book
  let $a := $b//author
  where count($a)>=2
  return <libro>
    <titolo> {$b/title/text()} </titolo>
    </libro>
}
</risultato>
```

che restituisce il titolo dei libri scritti da almeno due autori. Una scrittura più veloce ma meno leggibile potrebbe essere evitare l'utilizzo di \$a ed avere:

$$\text{count}(\$b//\text{author}) \geq 2$$

Restituire un documento la cui radice ha un attributo numTitoli che contiene il numero di elementi <libro> presenti nel document. La radice contiene inoltre una lista di elementi titolo, uno per ogni libro:

```
<risultato numTitoli="{count(document(...) // book)}">
{
  for $t in document(...) // title
  return <titolo>
    {$t/text()}
  </titolo>
}
</risultato>
```

Restituire i libri scritti da "Dan Suciù", si noti che un libro è scritto da una persona quando è all'interno di una lista degli autori di un libro:

```
<risultato>
{
    for $b in document(...) // book[publisher="Dan Suciù"]
    where $b/author/last="Suciù" and $b/author/first="Dan"
    return $b/title
}
</risultato>
```

questa query è sbagliata in quanto ammette un libro scritto ad esempio da Dan Peterson e Pasqualino Suciù, si corregge nel seguente modo:

```
<risultato>
{
    for $b in document(...) // book[author[last=Suciù][first=Dan]]
    for $a in $b/title
}
</risultato>
```

o anche:

```
<risultato>
{
    for $b in document(...) // book
    for $a in $b/author
    where $a/first="Dan" and $a/last="suciù"
    return $b/title
}
</risultato>
```

questa è corretta in quando la condizione della where è applicata allo stesso autore.



Per ogni publisher si vogliono sapere i libri, il DTD avrà struttura:

```
<!ELEMENT resutl (publisher+)>
<!ELEMENT publisher (name, listOfBooks)>
<!ELEMENT listOfBooks (title+)>
<!ELEMENT title(#PCDATA)>
<!ELEMENT name (#PCDATA)>
```

si procede alla scrittura in XQuery:

```
<risultato>
{
  for $p in distinct-values (document(...)//publisher)
  return <publisher>
    <name> {$p/text()} </name>
    <listOfBooks>
      {
        document(...)//book[publisher=$p]/title
      }
    </listOfBooks>
  </publisher>
}
</risultato>
```

Il comando "distinct-values" permette di selezionare tutti figli "publisher" diversi tra loro.

## 10 Transazione

Ciò che distingue un sistema basato su DBMS e un sistema di gestione di dati è il fatto che il DBMS gestisce le Transazioni.

Una transazione è un programma in esecuzione che accede alla base di dati per estrarre i dati ed eventualmente modificarne lo stato.

Si supponga di avere una base di dati con la seguente configurazione:

Fattura(numero,anno,cliente,data)  
Cliente(CF,nome,...)  
Composizione(nFattura,aFattura,prodotto,qta, prezzoU)  
Prodotto(codice,nome,...)

Per inserire una fattura:

- Inserire 1 tupla in fattura
- Forse fare l'inserimento in cliente
- Forse fare n inserimenti in prodotti
- Fare n inserimenti in Composizione

L'inserimento di una fattura è una Transazione, in quanto anche se è divisa in più statement di basso livello, concettualmente è un'unica operazione che o è svolta interamente o non deve essere eseguita.

Nel database dato un insieme di operazioni inserire una COMMIT indica che queste devono essere prese come corpo di una transazione:

INSERT  
UPDATE  
.  
.  
COMMIT

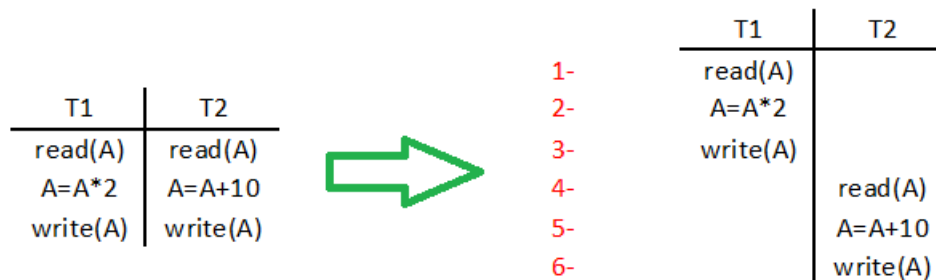
Se la transazione per qualsiasi motivo non viene completata è necessario ripristinare l'integrità della base di dati questo è detto ROLL BACK.

Le proprietà garantite da un sistema transazionale sono 4 e vengono sintetizzate nella sigla ACID:

- Atomicità: ha successo se tutti gli statement della transazione hanno successo
- Consistenza: una transazione agisce sullo stato consistente di una base di dati e lascia la base di dati in uno stato consistente
- Isolamento: più transazioni che concorrentemente lavorano sulla base di dati sono isolate tra loro, viene quindi gestita la concorrenza delle modifiche
- Durabilità: dopo il commit, se questa ha successo, lo stato del DBMS è definitivo e dura nel tempo

Dato un insieme di transazioni, dare un ordine di esecuzione è detto SCHEDULING SERIALE, se si ha uno scheduling non seriale si ha il rischio di race condition.

Lo SCHEDULE è l'assegnazione di un numero ordinale ad ogni operazione di un set di transazioni:



Uno schedule S è SERIALIZZABILE se esiste S' sulle stesse transizioni di S tale che:

$$S \equiv S'$$

ossia  $\forall$  istanza del database possibile:

$$S(D) = S'(D)$$

## 10.1 Isolamento

Si considerano due transazioni:

T1	T2
read(A)	read(A)
A=A+10	tmp=0,2*A
write(A)	A=A+tmp
read(B)	write(A)
B=B-10	read(B)
write(B)	B=B-tmp
	write B

I due possibili scheduler seriali sono:

$$T_1, T_2 \qquad T_2, T_1$$

Partendo da A=100 e B=100:

- $T_1, T_2 \rightarrow A = 132, B = 68$
- $T_2, T_1 \rightarrow A = 130, B = 70$

E si considera lo schedule:

	T1	T2
1-	read(A)	
2-	A=A+10	
3-	write(A)	
4-		read(A)
5-		tmp=0,2*A
6-		A=A+tmp
7-		write(A)
8-	read(B)	
9-	B=B-10	
10-	write(B)	
11-		read(B)
12-		B=B-tmp
13-		write B

Lo schedule non è seriale in quando le op di T1 sono bloccate da T2 e viceversa, ma è serializzabile in quanto è pari all'esecuzione sequenziale di T1,T2.

## 10.2 Schedule Serializable

Si indichi con  $S$  l'insieme di tutti gli schedule serializzabili, il database solitamente genera dato un insieme di transazioni schedule che stanno nelle classi:

- Conflict-Serializable
- View-Serializable

### 10.2.1 Conflict Serializable

Uno schedule  $S$  è CONFLICT SERIALIZABLE se è conflict equivalent ad uno schedule seriale  $S'$ :

$$S \equiv_C S'$$

ovvero se esiste una sequenza di equazioni non in conflitto tramite le quali si ottiene  $S'$  da  $S$ .

Non formalmente uno scheduler è conflict serializable quando dallo schedule scritto è possibile ottenere uno scheduler equivalente seriale, facendo uno switch tra operazioni non in conflitto. Considerando lo scheduler preso in esempio precedentemente questo verifica la condizione, in quanto effettuando le sostituzioni si ottiene  $T_1T_2$ :

	T1	T2
1-	read(A)	
2-	A=A+10	
3-	write(A)	
4-	read(B)	
5-	B=B-10	↓
6-	write(B)	
7-		read(A)
8-	↑	tmp=0,2*A
9-		A=A+tmp
10-		write(A)
11-		read(B)
12-		B=B-tmp
13-		write B

Si noti, dal grafico insiemistico precedente, che se uno schema è CS è serializzabile. Inoltre due operazioni sono in conflitto quando avvengono sulla stessa risorsa e almeno una è una write.

Si supponga di avere un nuovo schedule:

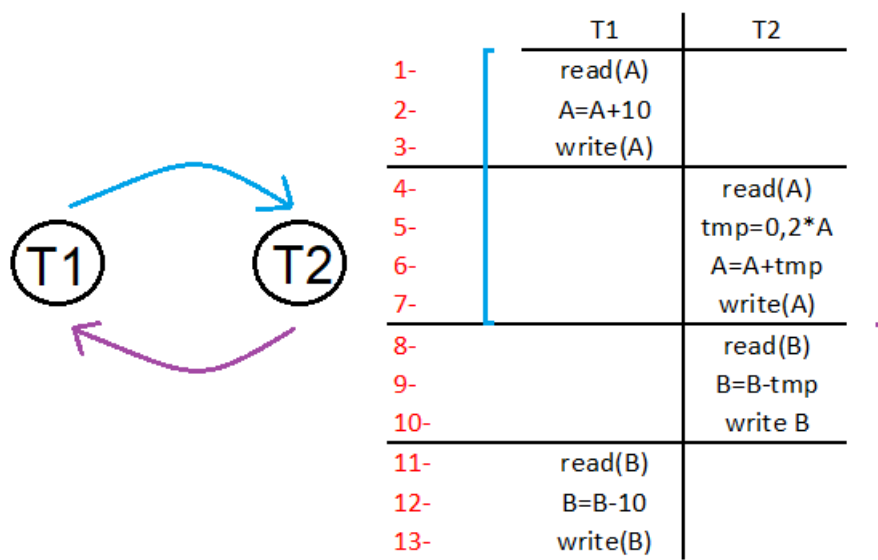
	T1	T2
1-	read(A)	
2-	A=A+10	
3-	write(A)	
4-		read(A)
5-		tmp=0,2*A
6-		A=A+tmp
7-		write(A)
8-		read(B)
9-		B=B-tmp
10-		write B
11-	read(B)	
12-	B=B-10	
13-	write(B)	

questo è:

$$\equiv T_1T_2$$

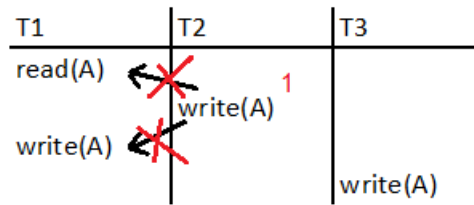
ma non è CS in quanto non è possibile effettuare alcuno scambio tra operazioni in conflitto, per ottenere la seq.  $T_1T_2$ .

Per capire se uno scheduler è CS, si prendono le transazioni e si disegna il GRAFO DELLE PRECEDENZE. I nodi sono dati dalle transazioni e ogni arco dalla transazione X alla transazione Y, indica che X e Y hanno una coppia di operazioni in conflitto tale che la prima si trova in X e la seconda in Y. Facendo riferimento all'esempio precedente:



Quando si ha un ciclo nell'arco, lo scheduler non è CS.

Si considera il seguente scheduler, indicando solo operazioni di read e write:



non è CS in quanto l'operazione ① non può essere scambiata con la precedente/successiva. Inoltre il valore finale di A è gestito solamente da T3, che non fa alcuna read ma va a scrivere un nuovo valore di A, per cui le transazioni T1 e T2 sono superflue. La transazione T3 è detta BLIND WRITE, scrittura ceca, in quanto non tengono in considerazione le transazioni precedenti.

### 10.2.2 View Serializable

Uno schedule S è VIEW SERIALIZABLE se è view Equivalent ad uno scheduler S'

$$S \equiv_V S'$$

Due schedule S, S' sono VIEW EQUIVALENT se:

1.  $\forall$  risorsa X, le transazioni che leggono il valore iniziale di X in S leggono tale valore anche in S' (e viceversa)
2.  $\forall$  risorsa X, se  $T_i$  legge il valore di X scritto da  $T_j$  in S, allora  $T_i$  legge il valore di X scritto da  $T_j$  in S'
3.  $\forall$  risorsa X, la transazione (se esiste) che scrive il valore di X in S scrive il valore finale di X in S' e viceversa.

Prendendo come riferimento lo scheduler precedente e la sequenza  $T_1T_2T_3$  questa non è VS in quanto già la 1° proprietà non è rispettata (nello scheduler lo leggono T1 e T2, mentre nella sequenza delle transazioni T2 legge il valore scritto da T1).

Si supponga ora di avere più transazioni interlacciate tra loro:


T1	T2
.	.
.	.
write(A)	read(A)
	write(A)
commit	.
	commit

lo schedule è CS per cui è serializzabile, ma può succedere (invertendo l'esecuzione di T1 e T2) che prima di fare commit avvenga un errore (questo perché T2 legge un valore scritto da T1, che però fa commit dopo T2), per cui sarà necessario fare ROLL BACK.

Nasce il problema della RECUPERABILITY (recuperabilità) ovvero garantire che in caso di errore sia sempre possibile ripristinare UNO stato consistente. Questo non vuol dire ripristinare lo stato prima dello schedule partisse. Il problema di Irrecuperabilità è strettamente legato alle commit.

Data una coppia di transazioni tale che la prima legge un valore scritto da un'altra, affinché ci sia recuperabilità, la prima transazione a fare commit deve essere quella che scrive.

Supponendo che nel momento in cui viene fatta la commit di T1 si crei un errore è necessario fare roll back, per cui viene annullato sia le operazioni fatte da T1 che da T2 in quanto queste dipendono dalla 1° transazione. Pensando di avere n transazioni, si viene a creare un Roll Back A CASCATA. Nelle basi di dati si cerca di ottenere scheduler privi di effetti a cascata, questo avviene se si garantisce che la transazione legga una risorsa già consolidata, per cui che T1 venga "committata" prima:

T1	T2
.	.
.	.
write(A)	
commit	
 commit	read(A)
	write(A)
	commit



### 10.3 Lucchetti

Molti DBMS, per rendere gli schedule Conflict Serializable, utilizzano due tipi di lucchetti:

- **Lucchetti idonei alla scrittura:** detto anche lock-X (eXclusive)
- **Lucchetti idonei alla lettura:** detto anche lock-S (Shared) , questo può essere aperto da più chiavi.

Se il lock-S è attivo:

- Tutte le richieste di lock-X vengono messe in attesa
- Tutte le richieste di lock-S vengono accettate

Se il lock-X è attivo:

- Tutte le richieste di lock-S sono messe in attesa
- Tutte le richieste di lock-X sono messe in attesa

I lucchetti consentono un maggior parallelismo, in quanto è possibile passare da una transazione all'altra senza problemi.

Si consideri l'esempio seguente, senza uso dei lucchetti:

	T1	T2
1)	read(A)	
2)	A=A+10	
3)		read(A)
4)		A=A+20
5)	write(A)	
6)		write(A)

Lo schedule non è serializzabile in quanto a seconda dell'ordine di esecuzione, si ottengono risultati diversi.

Si implementino ora i lucchetti:

	T1	T2
1)	Lock-X(A)	
2)	read(A)	
3)	A=A+10	
4)		Lock-X(A) not granted
5)	write(A)	
6)	unlock(A)	
7)		Lock-X(A) granted
8)		read(A)
9)		A=A+20
10)		write()
11)		unlock(A)

Viene garantita la serializzabilità, ma questo non è sempre detto, per dimostrarlo si fornisce un controesempio:

	T1	T2
1)	Lock-X(A)	
2)	read(A)	
3)	A=A+10	
4)	unlock-X(A)	
5)		lock-X(A)
6)		read(A)
7)		A=A*2
8)		write(A)
9)		unlock-X(A)
10)		lock-X(B)
11)		read(B)
12)		B=2*B
13)		write(B)
14)		unlock-X(B)

I lucchetti vengono richiesti in modo corretto, ma la sua esecuzione non corrisponde a nessuno dei due possibili schemi di esecuzione  $T_1, T_2$  o  $T_2, T_1$ .

### 10.3.1 2 Phase Locking

Affinche venga garantito isolamento è necessario che una transazione  $T_i$  una volta preso il lucchetto della risorsa X, non lo rilascia se deve richiederne altri su altre risorse.

Questa caratteristica è verificata nel protocollo 2PL, che si compone di due fasi:

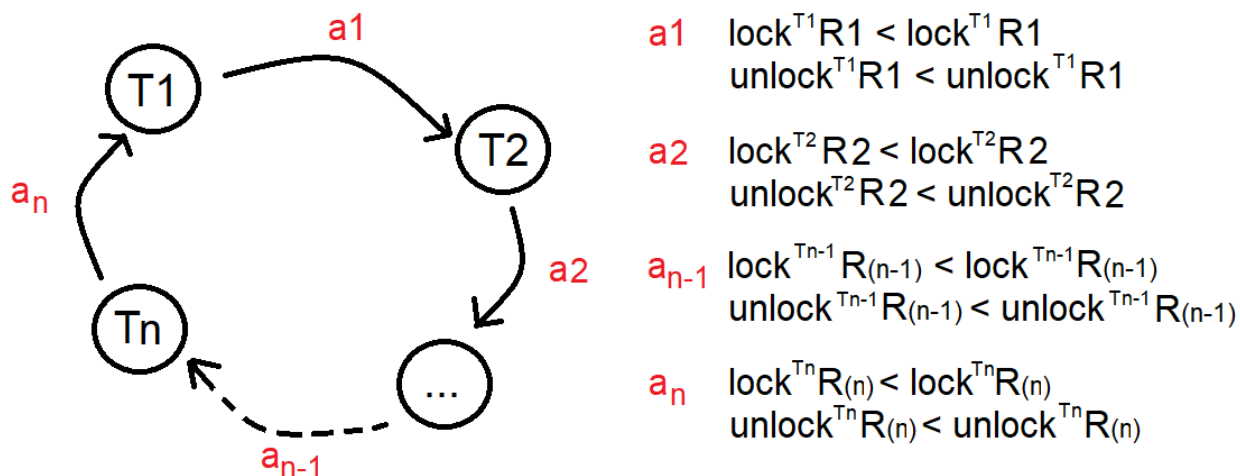
- **Growing** → cresce il numero di lucchetti posseduto da una transazione
- **Shrinking** → vengono rilasciati tutti i lucchetti

#### Dimostrazione di CS con 2PL

Si considerino n transazione che aderiscono dalla 2PL.

Sia S uno schedule generato secondo la semantica dei lucchetti

S per ASSURDO non è CS ed è caratterizzato dal grafo delle transazioni:



Ne esce fuori che:

$\text{unlock}(n)$  di T1 precede la  $\text{lock}(R_1)$  di T1

Questo non è possibile in quanto nel protocollo 2PL, ma in generale, non può avvenire una unlock prima di una lock per cui si è all'ASSURDO.

Il protocollo 2PL non garantisce la recuperabilità in quanto non chiarisce nulla riguardo la posizione della commit, nascono due varianti:

- **Strict 2PL**

I lucchetti esclusivi vengono rilasciati dal comando commit. Questa variabile del 2PL serve per garantire la recuperabilità e la cascadeless. Nella pratica non viene utilizzato tale protocollo nei database.

- **Strong Strict 2PL**

In questo caso tutti i lucchetti o meno vengono rilasciati nella committed. Per un motivo di praticità, nessuna transazione dovrà scrivere unlock. Ovviamente più vincoli mettiamo più si riduce di grado di libertà dello schedule, nel senso che ci sarebbe schedule, ragionevoli, che non potranno mai essere ragionati perché non rispetterebbero lo Strong Strict 2PL.