

# Basi Di Dati

Anno 2022/2023

A CURA DI:  
Erica Corda  
Matteo Dessì  
Leonardo Dessì  
Simone Giuffrida

# DBMS E MODELLO RELAZIONALE

Per iniziare a trattare dei database, che sono un [insieme di dati](#), ci deve essere chiara la distinzione fra [informazioni](#) e [dati](#):

- L'[informazione](#) è una notizia, una nozione, un elemento, cioè un qualsiasi cosa che può essere interpretato. Consente di avere conoscenza più o meno esatta di fatti, situazioni, modi di essere;
- Il [dato](#) è una temperatura, un'età, cioè ciò che è immediatamente presente alla conoscenza, prima di ogni elaborazione. In informatica sono gli elementi di informazione costituiti da simboli che devono essere elaborati.

Nelle attività umane, le informazioni vengono gestite in forme diverse:

- idee informali;
- linguaggio naturale (scritto o parlato);
- disegni, grafici, schemi;
- sistemi informatici complessi

E su vari supporti come la mente umana, la carta, e i dispositivi elettronici.

I [DBMS](#) ( DataBase Management System), in particolare quelli [relazionali](#), sono specializzati nel trattare i dati, non le informazioni.

Le Basi di Dati si possono interpretare in due modi:

1. Come un insieme organizzato di dati utilizzati per il supporto allo svolgimento di attività (azienda, ufficio, persone,...);
2. Come un insieme di dati gestito da un Database Management System (DBMS), cioè, un software specializzato nella memorizzazione e interrogazione di dati.

I DMBS devono gestire una grande quantità di dati, questi ultimi devono essere [mantenuti per molto tempo](#); perciò, viene utilizzata una memoria di massa in cui memorizzarli e non una RAM.

Devono [persistere nel tempo](#), in quanto hanno un tempo di vita indipendente dalle singole esecuzioni dei programmi che le utilizzano e deve essere possibile [condividerli](#), questo ci porta a dover stabilire dei meccanismi di autorizzazione e di controllo della concorrenza.

Devono garantire principalmente 3 caratteristiche:

- [Privatezza](#): i dati devono essere privati, ciò significa che bisogna predisporre delle modalità d'accesso con cui vengono attribuiti diversi diritti agli utenti che ci vogliono accedere;
- [Affidabilità](#): bisogna garantire che i dati non vengano persi, quindi garantire la resistenza a malfunzionamenti hardware e/o software;
- [Efficienza](#): questi sistemi devono essere efficienti, quindi bisogna cercare di utilizzare al meglio le risorse come lo spazio in memoria e il tempo (fornire la risposta nel minor tempo possibile).

Esistono due principali tipi di [modelli](#):

- [Concettuali](#): permettono di rappresentare i dati in maniera indipendente da ogni sistema. cercano di descrivere i concetti del mondo reale e sono utilizzati nelle fasi preliminari di progettazione. Il più diffuso è il modello [Entity-Relationship \(ER\)](#);
- [Logico](#): è un insieme di costrutti utilizzati per organizzare i dati di interesse, la componente fondamentale sono i meccanismi di strutturazione. Il principale modello logico è il modello relazionale che prevede il [costrutto relazione](#) che permette di definire un insieme di record omogenei.

In ogni base di dati esistono:



- Lo **schema** che descrive la sua struttura, che generalmente è statica e quindi non cambia nel tempo;
- L'**istanza** cioè l'insiemi dei campi di una considerata riga, quindi sono dei valori che possono cambiare.

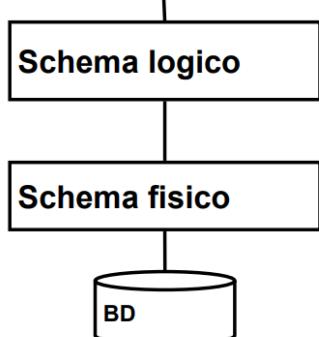
Tabella “Corsi” Schema della base di dati

Insegnamento	Docente	Aula	Ora
Analisi matem. I	Luigi Neri	N1	8:00
Basi di dati	Piero Rossi	N2	9:45
Chimica	Nicola Mori	N1	9:45
Fisica I	Mario Bruni	N1	11:45
Fisica II	Mario Bruni	N3	9:45
Sistemi inform.	Piero Rossi	N3	8:00

Istanza della base di dati

## Architettura di un DBMS

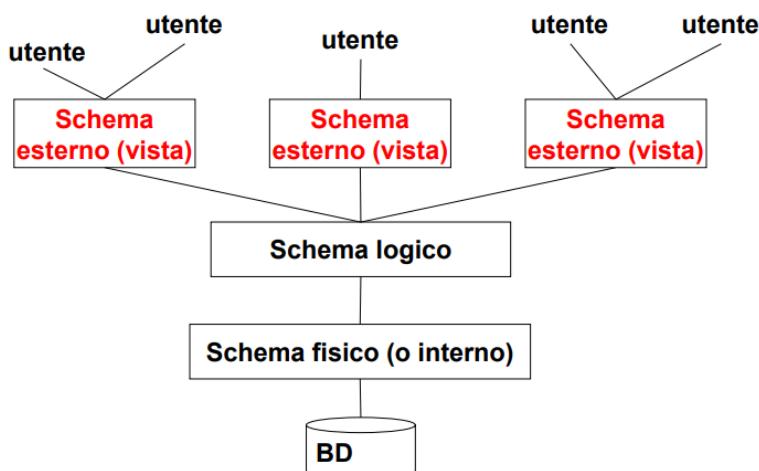
### Utente e programmi



Il progettista definisce lo **schema concettuale**, che viene tradotto nello **schema logico** (tabelle), implementate nel DBMS relazionale tramite strutture dati e algoritmi opportuni (livello fisico). L'utente e i programmi interagiscono solo con lo schema logico.

Esiste un **architettura standard a tre livelli** per DBMS in cui si hanno:

- Lo **schema esterno**: cioè la “vista” parziale o derivata di una parte della base di dati in un modello logico;
- Lo **schema logico**: cioè la descrizione dell'intera base di dati nel modello logico “principale” del DBMS;
- Lo **schema fisico**: cioè la rappresentazione dello schema logico per mezzo di strutture fisiche di memorizzazione.



Una conseguenza della articolazione in livelli è **l'indipendenza dei dati**, l'accesso a quest'ultimi avviene solo tramite il livello esterno che può coincidere con il livello logico.

Si hanno due tipi di indipendenza:

- **Fisico**: il livello logico e quello esterno sono indipendenti da quello fisico. Una relazione è utilizzata nello stesso modo qualunque sia la sua realizzazione fisica, la realizzazione fisica può cambiare senza che debbano essere modificati i programmi;
- **Logico**: il livello esterno è indipendente da quello logico. Aggiunte o modifiche alle viste (livello esterno) non richiedono modifiche al livello logico. Le modifiche allo schema logico che lasciano inalterato lo schema esterno sono trasparenti.

Per le basi di dati si utilizzano diversi linguaggi, in particolare per le basi di dati relazionali abbiamo diversi linguaggi e interfacce:

- il **linguaggio SQL**, il quale è un linguaggio testuale interattivo in quanto ci fornisce delle risposte;
- interagiamo con il DBMS con i comandi SQL che sono immersi in un **linguaggio ospite** (Python, java, C,...);
- le **interfacce amichevoli** come interfacce web, che sono prive di linguaggio testuale, in cui l'utente non sa com'è strutturato il database che si trova dietro, ma può comunque mandare dei dati al DBMS.

Abbiamo due **classi di linguaggi** per i DBMS:

- **Data definition language (DDL)**: serve per definire lo schema del linguaggio, quindi per la definizione e la modifica di schemi (logici (tabelle), esterni (viste → tabelle vere e proprie), fisici (strutture dati e algoritmi)) e altre operazioni generali;
- **Data manipulation language (DML)**: serve per l'interrogazione e l'aggiornamento (inserimento, cancellazione, modifica, di righe nelle tabelle) di istanze nelle basi di dati.

Corsi			Aule		
Corso	Docente	Aula	Nome	Edificio	Piano
Basi di dati	Rossi	DS3	DS1	OMI	Terra
Sistemi	Neri	N3	N3	OMI	Terra
Reti	Bruni	N3	G	Pincherle	Primo
Controlli	Bruni	G			

- "Trovare i corsi tenuti in aule a piano terra"

```
SELECT Corso, Aula, Piano
FROM Aule, Corsi
WHERE Nome = Aula
AND Piano = 'Terra'
```

Corso	Aula	Piano
Sistemi	N3	Terra
Reti	N3	Terra

Un'operazione DDL sullo schema è la creazione di una tabella, che con il linguaggio SQL si crea così:

```
CREATE TABLE orario (
    insegnamento  CHAR(20) ,
    docente        CHAR(20) ,
    aula           CHAR(4) ,
    ora            CHAR(5) )
```

## DOMANDE VERO E FALSO

1. L'indipendenza dei dati permette di scrivere programmi senza conoscere le strutture fisiche dei dati. **V**
2. L'indipendenza dei dati permette di modificare le strutture fisiche dei dati senza dover modificare i programmi che accedono alla base di dati. **V**
3. L'indipendenza dei dati permette di scrivere programmi conoscendo solo lo schema concettuale della base di dati. **F** perché per scrivere i programmi bisogna conoscere lo schema logico non quello concettuale;
4. La distinzione fra DDL e DML corrisponde alla distinzione fra schema e istanza. **V**
5. Le istruzioni DML permettono di modificare la base di dati ma non di interrogarla. **F**
6. Le istruzioni DDL permettono di specificare la struttura della base di dati ma non di modificarla. **F**
7. Non esistono linguaggi che includono sia istruzioni DDL sia istruzioni DML. **F**

All'interno del DBMS si hanno diversi ruoli:

- Progettisti e sviluppatori, che lavorano a livello fisico;
- Progettisti e amministratori della base di dati (**DBA**), che lavorano a livello logico, esterno e concettuale e un po' anche a livello fisico;
- Analisti, progettisti e sviluppatori di applicazioni, che lavorano a livello esterno;
- Utenti, non accedono direttamente al db ma tramite interfacce.

Il **Database administrator** (DBA) è una persona o un gruppo di persone responsabile del controllo centralizzato e della gestione del sistema, delle prestazioni, dell'affidabilità, delle autorizzazioni. Le funzioni del DBA includono quelle di progettazione, anche se in progetti complessi ci possono essere distinzioni.

## Il modello relazionale

Il modello relazionale è il **modello logico** attualmente più diffuso, nasce negli anni 70, ma diventa popolare all'inizio degli anni 80. Si chiama relazionale perché si basa sul **concetto matematico di relazione**, la quale è una naturale rappresentazione grazie all'utilizzo di tabelle.

La relazione può essere interpretata secondo tre diverse accezioni:

1. Relazione matematica: come nella teoria degli insiemi;
2. Relazione secondo il modello relazionale dei dati: si intende una relazione come una tabella;
3. Relazione (dall'inglese relationship) tra entità astratte nel modello Entity-Relationship (noi le chiameremo "associazioni"), che mette in relazione istanze di tabelle diverse.

## Relazione matematica, esempio

a	x	D1={a,b}
a	y	D2={x,y,z}
a	z	
b	x	Il prodotto cartesiano $D1 \times D2 = \{(a,x), (a,y), (a,z), (b,x), (b,y), (b,z)\}$
b	y	
b	z	Una relazione r è un sottoinsieme del prodotto cartesiano, dove D1 e D2 sono detti i domini della relazione.
a	x	
a	z	Una relazione matematica è un insieme di n-uple i cui elementi (d1, d2, ...) sono ordinati
b	z	→ (d1, ..., dn) tali che d1 ∈ D1, ..., dn ∈ Dn.

Siccome la relazione matematica è un insieme:

- non c'è ordinamento fra le n-uple (le n-uple sono le righe, fra le colonne si vi è ordinamento);
- le n-uple ("tuple") sono distinte: un insieme non può contenere elementi ripetuti;
- ciascuna tupla è ordinata: l' i-esimo valore proviene dall' i-esimo dominio.

## *Partite* $\subseteq$ *String* $\times$ *String* $\times$ *Int* $\times$ *Int*

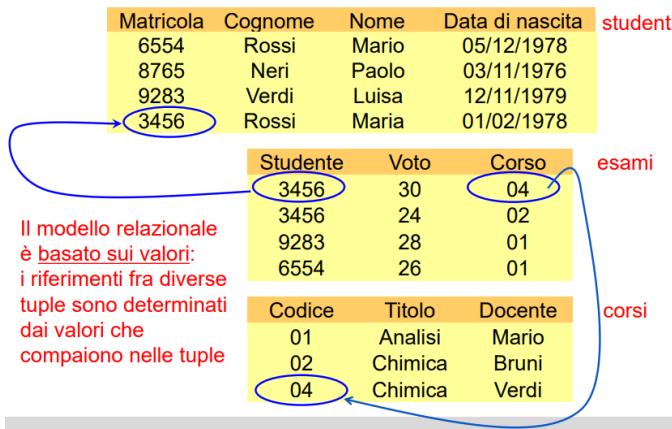
Juve	Lazio	3	1
Lazio	Milan	2	0
Juve	Roma	0	2
Roma	Milan	0	1

Ciascuno dei **domini** ha due ruoli diversi, distinguibili attraverso la posizione → la struttura della relazione matematica è **posizionale**.

Negli **RDBMS** (Relational DBMS) la struttura è **NON posizionale**; quindi, a ciascun dominio si associa un nome unico nella tabella (**attributo**), che ne descrive il "ruolo":

Casa	Fuori	RetiCasa	RetiFuori
Juve	Lazio	3	1
Lazio	Milan	2	0
Juve	Roma	0	2
Roma	Milan	0	1

Nel modello relazionale, in una tabella che rappresenta una relazione, l'ordinamento tra le righe è **irrilevante** e anche l'ordinamento tra le colonne lo è. Il modello relazione è **basato su valori**, ciò vuol dire che anche i riferimenti fra dati in strutture (relazioni) diverse sono rappresentati per mezzo dei valori stessi, è **gerarchico e reticolare**, quindi utilizza riferimenti esplicativi (puntatori) fra record.



Per definire uno **schema** di una relazione bisogna attribuirli un **nome** che poi diventerà la tabella nel db e di seguito al nome bisogna specificare un **insieme di attributi**  $A_1 \dots A_n$  che rappresentano le colonne della tabella. Ogni **attributo** A ha un proprio **dominio** cioè il tipo di dato. Lo schema delle basi di dati è un **insieme di schemi di relazioni** cioè le varie tabelle che compongono il db.

## Esercizio

Considerare le informazioni per la gestione delle presenze di pesci nelle vasche di un acquario civico. Ogni tipo di pesce ha un nome univoco (es. Piranha), e se ne conosce la famiglia (ciclidi, caracidi, ecc). Ogni vasca ha un nome univoco, e se ne conosce la sala, il piano e la capienza in litri. Si vuole tenere traccia delle presenze e della numerosità dei pesci nelle vasche. Definire uno schema logico per rappresentare queste informazioni, individuando opportuni domini per i vari attributi e mostrarne un'istanza in forma tabellare.

## Schema logico (relazionale)

- Pesce ( NomeP, Famiglia )
- Vasca ( NomeV, Sala, Piano, Capienza )
- Presenze ( NomeP, NomeV, Quantità )

PESCE	NomeP	Famiglia
	Scalare	Ciclidi
	Cardinale	Caracidi

VASCA	NomeV	Sala	Piano	Capienza
	SudAmerica	S3	1	2000
	Messico	S3	2	800

PRESENZE	NomeP	NomeV	Quantità
	Scalare	SudAmerica	12
	Scalare	Messico	4
	Cardinale	Messico	70

In alcuni casi può capitare di avere delle [basi di dati incomplete](#), in cui non conosciamo ancora un determinato dato, come possiamo comportarci?

NomeP	NomeV	Quantità
Scalare	Sudamerica	12
Scalare	Messico	
Cardinale	Messico	70

Non conviene usare valori del dominio (0, stringa vuota, "???", ...):

- potrebbero non esistere valori “non utilizzati”;
- valori “non utilizzati” potrebbero diventare significativi;
- in fase di utilizzo (nei programmi) sarebbe necessario ogni volta tener conto del “significato” di questi valori.

La soluzione è utilizzare [NULL](#) che denota l'assenza di un valore del dominio, non è un valore vero e proprio, ma rappresenta la mancanza di un valore per quell'attributo. Rappresenta un valore inesistente oppure sconosciuto.

Può identificare univocamente le tuple, quindi consente di evitare ambiguità, in generale è necessario garantire che ogni tupla sia identificabile univocamente, indipendentemente dalle modifiche che verranno apportate ai dati. L'identificazione della tupla permette di correlare i dati in relazioni diverse (il modello relazionale è basato su valori).

Per far ciò utilizziamo una [superchiave](#) che è un insieme di attributi che identificano univocamente le tuple di una relazione. Formalmente:

- un insieme K di attributi è [superchiave](#) per la relazione R, se R non contiene due tuple distinte t1 e t2 con gli stessi valori per gli attributi in K;
- K è [chiave](#) per R se è una [superchiave minimale](#) per R (cioè non contiene un'altra superchiave)

### ESEMPIO

Se l'attributo NomeV è chiave di Vasca, significa che non posso avere due vasche con lo stesso nome. Se la chiave di Vasca è la coppia di attributi < NomeV, Piano>, significa che non posso avere due vasche con lo stesso nome sullo stesso piano (ma potrei averle su piani diversi!).

Matricola	Cognome	Nome	Corso	Nascita
27655	Rossi	Mario	Ing Inf	5/12/78
78763	Rossi	Mario	Ing Inf	3/11/76
65432	Neri	Piero	Ing Mecc	10/7/79
87654	Neri	Mario	Ing Inf	3/11/76
67653	Rossi	Piero	Ing Mecc	5/12/78

In questo esempio matricola è una chiave che è una superchiave che contiene un solo attributo e quindi è minimale. Se scegliessimo <Cognome, Nome, Nascita> anche questa è un'altra chiave, ed è anche un superchiave minimale.

Una chiave è sempre una superchiave ma non vale l'inverso, una superchiave non è detto che sia anche una chiave.

Una superchiave è una chiave se è una superchiave minimale ossia se non contiene altre superchiavi al suo interno. Se prima avessimo scelto <Matricola, Cognome, Nome> questa non è una superchiave minimale perché ci sono altre superchiavi al suo interno ossia la superchiave matricola. Quindi non è una chiave.

Il teorema di esistenza delle chiavi ci dice che:

- Ogni tabella ha sempre almeno una superchiave: nel modello relazionale una tabella non può contenere due tuple uguali. Pertanto, l'insieme degli attributi della tabella è sicuramente una superchiave. Da questo si deduce che ogni tabella e ogni schema di relazione ha almeno una superchiave.
- Ogni tabella ha almeno una chiave: ogni tabella ha una superchiave e ogni superchiave ha al suo interno almeno una superchiave minimale ossia una chiave. Pertanto, in una tabella (relazione) è sempre presente almeno una chiave.

Potrebbero esistere diverse chiavi (ad esempio, sia Matricola, che ), bisogna scegliere una chiave primaria che poi verrà usata per la tabella del database. In generale, si preferiscono chiavi composte dal minor numero di attributi.

Se la chiave fosse troppo complessa, come in questo caso:

Esami medici	Paziente	Esame	Esito	Medico	Data	ID
	p23	TAC	POS	03	3/1/16	1
	p5	ECG	NEG	01	2/1/16	2
	p5	ECG	NEG	01	1/1/16	3
	p7	TAC	NEG	04	2/1/16	4

Si crea una chiave artificiale cioè si aggiunge un attributo (ID) e gli assegno dei valori univoci (ad esempio, un contatore incrementale).

Soltanamente la chiave primaria viene identificata con la sottolineatura e permette quindi di identificare univocamente tutte le tuple di una tabella in modo rapido, semplice ed efficace, in una tabella possono esserci molte chiavi ma soltanto una è anche una chiave primaria. Tutte le altre chiavi sono dette chiavi secondarie.

## DOMANDE VERO E FALSO

1. Ogni attributo appartiene al massimo ad una chiave. F
2. Possono esistere attributi che non appartengono a nessuna chiave. V
3. Possono esistere attributi che non appartengono a nessuna superchiave. F perché ad esempio anche tutti gli attributi possono comporre una superchiave, se non è minimale.
4. Una chiave può essere sottoinsieme di un'altra chiave. F

5. Può esistere una chiave che coinvolge tutti gli attributi. **V**
6. Può succedere che esistano più chiavi e che una di esse coinvolga tutti gli attributi. **F** perché se esiste una chiave che contiene tutti gli attributi non può esistere nessun sottoinsieme che sia chiave;
7. Una relazione può avere due chiavi primarie. **F**
8. Una superchiave non minimale può essere sottoinsieme di una chiave primaria. **F** viceversa sarebbe stato corretto.

## Esercizio: cosa scegli come chiave?

Esami	Studente	Voto	Lode	Corso
	222222	30		01
	111111	30	e lode	02
	333333	27		03
	222222	24		04

Come chiave  
scelgo la coppia:  
**<Studente, Corso>**  
(lo stesso studente  
può dare più esami  
ma non per lo  
stesso corso)

Se scelgo come chiave Studente, non posso avere due esami fatti dallo stesso studente! (nemmeno se fatti per corsi diversi).

Se scelgo <Studente, Voto>, non ammetto che lo studente possa prendere lo stesso voto in 2 esami diversi.

In presenza di **valori nulli**, i valori della chiave non permettono:

- di identificare le tuple;
- di realizzare i riferimenti da altre relazioni.

Gli attributi che compongono la chiave primaria non possono assumere valori nulli!

## ESEMPIO basi di dati scorrette

Esami	Studente	Voto	Lode	Corso
	111111	32		01
	111111	30	e lode	02
	111111	18	e lode	03
	333333	24		04

Esistono istanze di basi di dati che non rappresentano informazioni possibili per l'applicazione di interesse.

In questo esempio possiamo notare come errori il voto 32, il voto 18 e lode, la matricola 333333 che non esiste come matricola nella tabella studenti, e la matricola 787643 associata a due studenti diversi.

Studenti	Matricola	Cognome	Nome
	111111	Rossi	Mario
	787643	Neri	Piero
	787643	Bianchi	Luca

Per definire delle regole con cui si stabilisce quali valori possono assumere le istanze, esistono i **vincoli di integrità**. Il vincolo di integrità è una proprietà che deve essere soddisfatta dalle istanze che rappresentano informazioni corrette per l'applicazione.

Un **vincolo** è una funzione booleana (un predicato) che associa ad ogni istanza il valore vero o falso.

Esistono diversi tipi di vincolo:

- **vincolo intra-relazionali**: definiti su unica tabella e sono valutati guardando quella tabella. Di cui esistono i **vincoli sui valori**, che coinvolgono un'unica colonna, e i **vincoli di tupla** che coinvolgono due colonne di una stessa tabella;
- **vincolo inter-relazionali**: definiti tra relazioni diverse, quindi bisogna controllare due tabelle diverse.

Alcuni tipi di vincoli (ma non tutti) sono supportati dai DBMS, possiamo specificarli nella nostra base di dati e il DBMS ne impedisce la violazione, questo permette di alleggerire il carico al programmatore.

Per i vincoli "non supportati", la responsabilità della verifica è del programmatore, o dell'utente, se non c'è interfaccia che controlli inserimenti, modifiche e cancellazioni.

Nei **vincoli inter-relazionali** le informazioni che si trovano in due relazioni devono essere correlate attraverso valori comuni, in particolare lo devono essere le chiavi primarie, quindi le **correlazioni** devono essere coerenti.

## ESEMPIO

Infrazioni

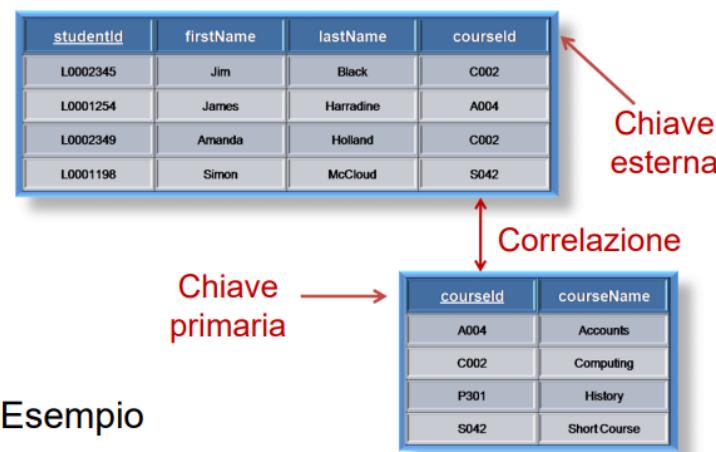
	Codice	Data	Vigile	Prov	Numero
Vigili	34321	1/2/95	3987	MI	39548K
	53524	4/3/95	3295	TO	E39548
	64521	5/4/96	3295	PR	839548
	73321	5/2/98	9345	PR	839548
	Matricola	Cognome	Nome		
3987	Rossi	Luca			
3295	Neri	Piero			
9345	Neri	Mario			
7543	Mori	Gino			

Infrazioni

	Codice	Data	Vigile	Prov	Numero
Auto	34321	1/2/95	3987	MI	39548K
	53524	4/3/95	3295	TO	E39548
	64521	5/4/96	3295	PR	839548
	73321	5/2/98	9345	PR	839548
	Prov	Numero	Cognome	Nome	
MI	39548K	Rossi	Mario		
TO	E39548	Rossi	Mario		
PR	839548	Neri	Luca		

Nel primo caso la **relazione è corretta** perché la matricola è corretta, ed esiste un unico vigile che non ha segnato alcuna infrazione (che può capitare), è ovviamente corretto che nella tabella dei vigili, il vigile con una determinata matricola compaia una sola volta, in quanto la matricola è la chiave primaria, invece nella tabella infrazioni può esistere uno stesso vigile che fa più multe, in questo caso la colonna vigile nella tabella infrazioni viene chiamata chiave esterna o **vincolo di integrità referenziale**.

Un **vincolo di integrità referenziale** (vincolo di chiave esterna, "foreign key") fra gli attributi X di una relazione R1 e un'altra relazione R2 impone ai valori su X in R1 di comparire come valori della chiave primaria di R2.



## Esercizio

Individuare le chiavi e i vincoli di integrità referenziale:

Chiavi primarie:

- ↗ “Cod” per la relazione PAZIENTI;
- ↗ “Paziente” e “Inizio” per la relazione RICOVERI;
- ↗ “Matr” per la relazione MEDICI;
- ↗ “Cod” per la relazione REPARTI.

La scelta fatta sulla relazione RICOVERI presume che un paziente possa essere ricoverato solo una volta nello stesso giorno.

I vincoli di integrità sono:

- ↗ Tra l'attributo “Paziente” in RICOVERI e “Cod” in PAZIENTI;
- ↗ Tra “Reparto” nella relazione RICOVERI e “Cod” nella relazione REPARTI ;
- ↗ Tra “Primario” in REPARTI e “Matr” nella relazione MEDICI;
- ↗ Tra “Reparto” in MEDICI e “Cod” in REPARTI.

PAZIENTI

Cod	Cognome	Nome
A102	Necchi	Luca
B372	Rossini	Piero
B543	Missoni	Nadia
B444	Missoni	Luigi
S555	Rossetti	Gino

REPARTI

Cod	Nome	Primario
A	Chirurgia	203
B	Medicina	574
C	Pediatria	530

RICOVERI

Paziente	Inizio	Fine	Reparto
A102	2/05/94	9/05/94	A
A102	2/12/94	2/01/95	A
S555	5/10/94	3/12/94	B
B444	1/12/94	2/01/95	B
S555	5/10/94	1/11/94	A

MEDICI

Matr	Cognome	Nome	Reparto
203	Neri	Piero	A
574	Bisi	Mario	B
431	Bargio	Sergio	B
530	Belli	Nicola	C
405	Mizzi	Nicola	A
201	Monti	Mario	A

88

Quando capita che viene eliminata una riga causando una **violazione**, il comportamento standard è quello di rifiutare l'operazione, ma esistono delle azioni che possono compensare queste violazioni come:

- L'eliminazione in cascata;
- L'introduzione di valori nulli o valori di default.

Una violazione può essere causata anche da un inserimento con un valore errato o inesistente, o con la modifica di un determinato campo.

Impiegati	Matricola	Cognome	Progetto
	34321	Rossi	IDEA
	64521	Verdi	<b>NULL</b>
	73032	Bianchi	IDEA

Impiegati	Matricola	Cognome	Progetto
	34321	Rossi	IDEA
	53524	Neri	<b>NULL</b>
	64521	Verdi	<b>NULL</b>
	73032	Bianchi	IDEA

Progetti	Codice	Inizio	Durata	Costo
	IDEA	01/2000	36	200
	BOH	09/2001	24	150

Progetti	Codice	Inizio	Durata	Costo
	IDEA	01/2000	36	200
	BOH	09/2001	24	150

## Esercizio

Definisci i vincoli di integrità referenziale:

- ↗ Studenti (Matricola, Cognome, Nome, DataNasc );
- ↗ Esami (Studente, Voto, Corso );
- ↗ Corsi (Codice, Titolo, Docente ).

I vincoli di integrità referenziali sono fra le tabella studenti ed esami, dove studente è la chiave esterna che corrisponde alla matricola (chiave primaria) dello studente, e fra le tabelle corsi e esami, dove corso è la chiave esterna che corrisponde al codice (chiave primaria) del corso.

- ↗ Cliente (codFisc, telefono, città );
- ↗ Acquisto (CF, IdProd, quantità, data );
- ↗ Prodotto (Id, nome, marca, prezzo ).

I vincoli di integrità referenziali sono fra le tabella cliente e acquisto, dove CF è la chiave esterna che corrisponde al codFisc(chiave primaria) di cliente, e fra prodotto e acquisto dove IdProd è la chiave esterna che corrisponde all'Id del prodotto.

## Algebra relazionale

L'**algebra** è una branca della matematica che studia come rappresentare un'informazione numerica tramite simboli e le regole per manipolarli; invece, l'**algebra relazionale** è un'algebra che viene utilizzata per modellare dati di **strutture relazionali** ed esprimere interrogazioni.

L'algebra relazionale è stata definita nei primi anni 70 da Edgar F. Codd ad IBM, è il fondamento teorico di linguaggi di interrogazione per database relazionali (tra cui SQL), e la base per implementare e ottimizzare i DBMS reali. Ci aiuta a capire come scrivere le **query**, gli operatori "di base" sono gli stessi, ma in SQL ne abbiamo di aggiuntivi (in SQL riusciamo a fare query più complesse).

I **linguaggi di interrogazione per basi di dati** relazionali sono di due tipi:

- **Dichiarativi**: specificano le proprietà del risultato (specificiamo "che cosa") → SQL;
- **Procedurali**: specificano le modalità di generazione del risultato ( specificiamo "come") → algebra relazionale.

L'algebra relazionale ci fornisce un insieme di operatori che ci permettono di:

- Eseguire operazioni su relazioni;
- Produrre nuove relazioni;
- E comporre altri operatori.

Gli operatori sono quelli:

- **Insiemistici**: unione, intersezione e differenza;
- Che lavorano sullo schema → **ridenominazione**;
- **Selezione e proiezione** (SELECT e PROJ);
- **Join**: per mettere assieme due relazioni.

Dato che le relazioni sono insiemi si possono applicare operatori insiemistici, i risultati devono essere a loro volta delle relazioni. È possibile applicare unione, intersezione, differenza solo a **relazioni definite sugli stessi attributi**.

## Unione

Laureati			Specialisti		
Matricola	Nome	Età	Matricola	Nome	Età
7274	Rossi	42	9297	Neri	33
7432	Neri	54	7432	Neri	54
9824	Verdi	45	9824	Verdi	45

Laureati  $\cup$  Specialisti

Matricola	Nome	Età
7274	Rossi	42
7432	Neri	54
9824	Verdi	45
9297	Neri	33

## Differenza

Laureati

Matricola	Nome	Età
7274	Rossi	42
7432	Neri	54
9824	Verdi	45

Specialisti

Matricola	Nome	Età
9297	Neri	33
7432	Neri	54
9824	Verdi	45

Laureati – Specialisti

Matricola	Nome	Età
7274	Rossi	42

## Intersezione

Laureati

Matricola	Nome	Età
7274	Rossi	42
7432	Neri	54
9824	Verdi	45

Specialisti

Matricola	Nome	Età
9297	Neri	33
7432	Neri	54
9824	Verdi	45

Laureati  $\cap$  Specialisti

Matricola	Nome	Età
7432	Neri	54
9824	Verdi	45

La [ridenominazione](#) permette di cambiare nome agli attributi e alle relazioni; quindi, modifica lo schema lasciando inalterate le istanze.

Paternità

Padre	Figlio
Adamo	Abele
Adamo	Caino
Abramo	Isacco

Paternità

Padre	Figlio
Adamo	Abele
Adamo	Caino
Abramo	Isacco

REN<sub>Genitore  $\leftarrow$  Padre</sub> (Paternità)

Genitore	Figlio
Adamo	Abele
Adamo	Caino
Abramo	Isacco

REN<sub>Genitore  $\leftarrow$  Padre</sub> (Paternità)

Genitore	Figlio
Adamo	Abele
Adamo	Caino
Abramo	Isacco

Maternità

Madre	Figlio
Eva	Abele
Eva	Set
Sara	Isacco

REN<sub>Genitore  $\leftarrow$  Madre</sub> (Maternità)

Genitore	Figlio
Eva	Abele
Eva	Set
Sara	Isacco

Impiegati	Cognome	Ufficio	Stipendio
Rossi	Roma	55	
Neri	Milano	64	

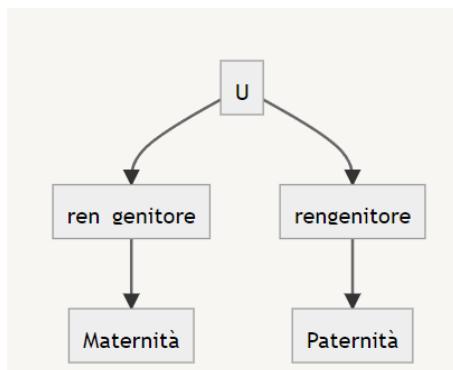
Operai	Cognome	Fabbrica	Salario
Bruni	Monza	45	
Verdi	Latina	55	

REN  $\text{Sede, Retribuzione} \leftarrow \text{Ufficio, Stipendio}$  (Impiegati)

REN  $\text{Sede, Retribuzione} \leftarrow \text{Fabbrica, Salario}$  (Operai)

Cognome	Sede	Retribuzione
Rossi	Roma	55
Neri	Milano	64
Bruni	Monza	45
Verdi	Latina	55

La **ridenominazione** è necessaria quando si ha la necessità di incrociare dati presi dalla stessa relazione.



La sintassi formale è indicata con la lettera **RHO**:

$$\rho_{R(A_1, \dots, A_n)}(E)$$

Data la relazione **E** (o una generica espressione) avente **n** attributi, rinomino **E** in **R** e i suoi attributi in **A1, A2, ..., An**.

Se si vuole rinominare solo pochi attributi si può usare la notazione più compatta:

REN  $\text{Sede, Retribuzione} \leftarrow \text{Fabbrica, Salario}$  (Operai)

Oppure:

$\rho_{\text{Sede, Retribuzione} \leftarrow \text{Fabbrica, Salario}}(\text{Operai})$

La **selezione** è un “filtro” sulle tuple che viene applicato in base a **condizioni** sui valori. Produce un risultato (anch’esso uno schema) che:

- Ha lo stesso schema dell’operando;
- Contiene un **sottoinsieme** delle tuple dell’operando (solamente quelle che rispettando la condizione);

### Impiegati

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

La selezione in questo esempio può esserci utile per trovare gli impiegati che:

- Guadagnano più di 50;
- Guadagnano più di 50 e lavorano a Milano;
- Hanno lo stesso nome della filiale presso cui lavorano.

La sintassi è questa:

## $\text{SEL}_{\text{Condizione}} (\text{Espressione})$

Dove la condizione è un'espressione booleana, la semantica di quest'operatore è che il risultato contiene le ennuple dell'operando che soddisfano la condizione.

### ESEMPI

- impiegati che guadagnano più di 50
  - impiegati che guadagnano più di 50 e lavorano a Milano

#### $\text{SEL}_{\text{Stipendio} > 50} (\text{Impiegati})$

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

#### $\text{SEL}_{\text{Stipendio} > 50 \text{ AND } \text{Filiale} = 'Milano'} (\text{Impiegati})$

Matricola	Cognome	Filiale	Stipendio
5998	Neri	Milano	64

Una sintassi alternativa è indicata con la lettera *sigma*:

$$\sigma_{\text{Stipendio} > 50 \wedge \text{Filiale} = 'Milano'} (\text{Impiegati})$$

- impiegati che hanno lo stesso nome della filiale presso cui lavorano

#### $\text{SEL}_{\text{Cognome} = \text{Filiale}} (\text{Impiegati})$

Matricola	Cognome	Filiale	Stipendio
9553	Milano	Milano	44

La selezione e la proiezione sono degli operatori ortogonali:

- La selezione applica un **filtro orizzontale** (righe);
- La proiezione applica un **filtro verticale** (colonne).

La **proiezione** produce un risultato che ha solo una parte degli attributi che si sono specificati, per questo motivo la relazione risultante sarà una nuova tabella che contiene solo alcune colonne della tabella di partenza.

#### Impiegati

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

- per tutti gli impiegati:
  - matricola e cognome
  - cognome e filiale

La sintassi è questa:

**PROJ** ListaAttributi (**Operando**)

Quello che ci restituirà sono le ennuple ottenute dopo aver applicato il filtro. Nella tabella sopra vogliamo sapere per tutti gli impiegati matricola e cognome; quindi, in questo caso la proiezione ci restituirà una tabella contenente due colonne.

- matricola e cognome di tutti gli impiegati
- cognome e filiale di tutti gli impiegati

Matricola	Cognome
7309	Neri
5998	Neri
9553	Rossi
5698	Rossi

Cognome	Filiale
Neri	Napoli
Neri	Milano
Rossi	Roma

**PROJ** Matricola, Cognome (**Impiegati**)

**PROJ** Cognome, Filiale (**Impiegati**)

La sintassi alternativa è indicata con il **pigreco**:

$\pi_{Cognome, Filiale}(Impiegati)$

## Esercizio

Consideriamo **PROJ<sub>K</sub>(R)**, se K è una superchiave di R, cioè che non esistono due tuple in R che hanno gli stessi valori per K, quante tuple contiene **PROJ<sub>K</sub>(R)**?

- può contenere meno tuple di R;
- può contenere più tuple di R;
- esattamente tante tuple quanto R**

Se facessi la proiezione della matricola e degli studenti, questa mi restituirà tutti gli studenti (tutte le loro matricole), se invece la facessi con il nome e il cognome, dato che non sono univoci, quelli con lo stesso nome e cognome vengono “messi assieme” e quindi la tabella risultante avrà meno valori. Quindi la risposta corretta è la C.

Combinando selezione e proiezione si possono estrarre informazioni interessanti da una relazione:

- matricola e cognome degli impiegati che guadagnano più di 50

Matricola	Cognome
7309	Rossi
5998	Neri
5698	Neri

**PROJ**Matricola, Cognome (**SEL**Stipendio > 50 (**Impiegati**))

Prima si esegue la **selezione** così otteniamo solo gli impiegati che hanno lo stipendio > 50 e successivamente dalla tabella risultante vengono estratti la matricola e il cognome con l'utilizzo della **proiezione**.

## Esercizio

Data la seguente relazione:

Persona(CF, Età, CittàNatale, CittàAttuale)

Si scriva l'espressione in algebra relazionale per rispondere alla seguente domanda:

- In quali città abitano dei 30enni?

$$PROJ_{Città' Attuale}(Sel_{Eta'=30}(Persona))$$

- In quali città è nato qualcuno ma non abita nessuno?

$$\rho_{città' \leftarrow Città' Natale} \prod_{Città' Natale} (Persona) - \rho_{città' \leftarrow Città' Attuale} \prod_{Città' Attuale} (Persona)$$

Si **rinomina** ( $\rho$ ) la colonna città Natale con il nome città e si fa la **proiezione** delle città in cui è nato qualcuno, si fa lo stesso con le città attuali, per ottenere il risultato si fa la **sottrazione** fra le varie città natali e le città attuali.

- Quali sono le città in cui è nato qualcuno e abita qualcuno?

$$\rho_{città' \leftarrow Città' Natale} \prod_{Città' Natale} (Persona) \cap \rho_{città' \leftarrow Città' Attuale} \prod_{Città' Attuale} (Persona)$$

Si fa l'**intersezione** fra le città natali ottenute con la **proiezione** e le città attuali, sempre ottenute con la **proiezione**.

- In quali città risiede qualche nativo?

$$PROJ_{Città' Natale}(Sel_{Città' Natale=Città' Attuale}(Persona))$$

Come prima cosa si fa la **selezione** delle tuple in cui la città attuale è uguale alla città nativa, successivamente si fa la **proiezione** sulle città natali, per ottenere le città in cui risiede qualche nativo.

- Quali sono tutte le città presenti nella base di dati?

$$\rho_{città' \leftarrow Città' Natale} \prod_{Città' Natale} (Persona) \cup \rho_{città' \leftarrow Città' Attuale} \prod_{Città' Attuale} (Persona)$$

Si fa l'**unione** fra le città natali ottenute con la **proiezione** e le città attuali, sempre ottenute con la **proiezione**, sempre considerando di fare la **ridenominazione** di città che sarà il campo che visualizzeremo.

- In quali città abitano solo anziani (età  $\geq 70$ )?

$$PROJ_{Città Attuale}(Persona) - PROJ_{Città Attuale}(Sel_{Eta<70}(Persona))$$

Si fa la **differenza** fra la **proiezione** delle città attuali e la **proiezione** delle città attuali in cui non ci sono anziani (con la **selezione**). Se noi avessimo solo fatto la Sel con età  $> 70$  avremo trovato solo le città in cui abita almeno un anziano, non le città in cui ci sono solo degli anziani.

**Combinando** selezione e proiezione, possiamo estrarre informazioni **da una relazione**, non possiamo però correlare informazioni presenti in relazioni diverse, né informazioni in enneuple diverse di una stessa relazione. Per far ciò si utilizza il **join**.

Il **join** è l'operatore che permette di correlare dati in relazioni diverse.

Ci sono tanti tipi di join, il primo che vediamo è il **join naturale**, il quale è un operatore binario che produce un risultato sull'unione degli attributi degli operandi con tuple costruite ciascuna a partire da una tupla di ognuno degli operandi.

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

Impiegati NAT_JOIN Reparti		
Impiegato	Reparto	Capo
Rossi	A	Mori
Neri	B	Bruni
Bianchi	B	Bruni

Ogni tupla (riga) contribuisce al risultato questo ci fornisce quindi un **join completo**, perché i reparti sono A e B, nella tabella Impiegati abbiamo come reparti A e B.

Impiegati		Reparti	
Impiegato	Reparto	Reparto	Capo
Rossi	A	B	Mori
Neri	B	C	Bruni
Bianchi	B		

Impiegati NAT_JOIN Reparti		
Impiegato	Reparto	Capo
Neri	B	Mori
Bianchi	B	Mori

Questo invece rappresenta un **join non completo** in quanto abbiamo il reparto C nella tabella Reparti, ma non esiste alcun reparto C assegnato ad un impiegato, ma esiste un reparto A che in realtà non esiste nella tabella Reparti.

Impiegati		Reparti	
Impiegato	Reparto	Reparto	Capo
Rossi	A	D	Mori
Neri	B	C	Bruni
Bianchi	B		

Impiegati NAT_JOIN Reparti		
Impiegato	Reparto	Capo

Questo invece rappresenta un **join vuoto**, in quanto nella tabella Reparti abbiamo i reparti D e C, nella tabella Impiegati invece abbiamo i reparti A e B, che non esistono nella tabella Reparti; quindi, il join ci restituirà una tabella vuota.

Impiegati		Reparti	
Impiegato	Reparto	Reparto	Capo
Rossi	B		Mori
Neri	B	B	Bruni

Impiegati NAT_JOIN Reparti		
Impiegato	Reparto	Capo
Rossi	B	Mori
Rossi	B	Bruni
Neri	B	Mori
Neri	B	Bruni

Questo rappresenta un **join completo con nxm ennuple**, che rappresenta una sorta di prodotto cartesiano.

La sintassi è:

### Impiegati NAT\_JOIN Reparti

Quella alternativa è indicata con il simbolo "bowtie", cioè papillon:

### *Impiegati* $\bowtie$ *Reparti*

- Il join naturale di R1 e R2 contiene un numero di ennuple compreso fra zero e il prodotto di  $|R_1|$  e  $|R_2|$ ;

$$0 \leq |R_1 \text{ JOIN } R_2| \leq |R_1| \times |R_2|$$

- Se il join coinvolge una chiave di R2 , allora il numero di ennuple è compreso fra zero e  $|R_1|$ ;

$$0 \leq |R_1 \text{ JOIN } R_2| \leq |R_1|$$

- Se il join coinvolge una chiave di R2 e un vincolo di integrità referenziale (da R1 a R2 ), allora il numero di ennuple è pari a  $|R_1|$ .

$$|R_1 \text{ JOIN } R_2| = |R_1|$$

Le  $| |$  rappresentano la **cardinalità**.

Il join ha una difficoltà principale:

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

Impiegato	Reparto	Capo
Neri	B	Mori
Bianchi	B	Mori

In questo particolare esempio alcune tuple non contribuiscono al risultato e quindi vengono "tagliate fuori", la soluzione a questo è utilizzare il **join esterno**, il quale estende con valori nulli le ennuple che verrebbero tagliate fuori da un join interno. Esistono tre versioni:

- **Sinistro**: mantiene tutte le ennuple del primo operando, estendendole con valori nulli, se necessario;

- **Destro**: mantiene tutte le ennuple del secondo operando, estendendole con valori nulli, se necessario
- **Completo**: mantiene tutte le ennuple di entrambi gli operandi estendendole con valori nulli, se necessario.

**Impiegati**

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

**Reparti**

Reparto	Capo
B	Mori
C	Bruni

**Impiegati**

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

**Reparti**

Reparto	Capo
B	Mori
C	Bruni

**Impiegati NAT\_JOIN LEFT Reparti**

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

**Impiegati NAT\_JOIN RIGHT Reparti**

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

**Impiegati**

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

**Reparti**

Reparto	Capo
B	Mori
C	Bruni

**Impiegati NAT\_JOIN FULL Reparti**

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

Un altro tipo di join è il **prodotto cartesiano** che viene definito come nella teoria degli insiemi, contiene sempre un numero di tuple pari al prodotto delle cardinalità degli operandi (le tuple sono tutte combinabili).

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

**Impiegati JOIN Reparti**

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

La sintassi è:

**Impiegati JOIN Reparti**

Che è equivalente al simbolo [cross](#) (o moltiplicazione):

## *Impiegati × Reparti*

L'ultimo join che vedremo è il [theta join](#). In generale il prodotto cartesiano ha senso quasi solo se seguito da una selezione, quest'operazione, quindi il prodotto cartesiano + la selezione viene chiamata theta join, e viene indicata con:

$$R_1 \text{ JOIN}_{\text{Condizione}} R_2$$

La condizione è spesso una [congiunzione](#) (con l'utilizzo dell'operatore AND) di [atomi di confronto](#) A1 & A2 dove [θ](#) (theta) è uno degli operatori di confronto (=, >, <, ...).

Se l'operatore di confronto nel theta-join è sempre l'uguaglianza (=) allora si parla di [equi-join](#).

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

Impiegati JOIN <sub>Reparto=Codice</sub> Reparti			
Impiegato	Reparto	Codice	Capo
Rossi	A	A	Mori
Neri	B	B	Bruni
Bianchi	B	B	Bruni

In questo esempio è come se stessimo usando il prodotto cartesiano che ci restituisce tutte le possibili combinazioni, e successivamente la selezione che ci "seleziona" solo i campi che hanno il reparto = al codice. Il join naturale in questo caso non si può utilizzare perché non c'è nessun attributo in comune, l'unico modo per farlo sarebbe applicare prima la ridenominazione in modo tale che venga cambiato il nome del codice in reparto.

La sintassi è:

$$\text{Impiegati JOIN}_{\text{Reparto}=\text{Codice}} \text{ Reparti}$$

Che è equivalente a:

$$\text{Impiegati } \bowtie_{\text{Reparto}=\text{Codice}} (\text{Reparti})$$

Possiamo esprimere il theta-join usando il prodotto cartesiano? Basta applicare al risultato del prodotto cartesiano la selezione.

Impiegati		Reparti	
Impiegato	Reparto	Codice	Capo
$\text{Impiegati JOIN}_{\text{Reparto}=\text{Codice}} \text{ Reparti}$			
$\text{SEL}_{\text{Reparto}=\text{Codice}}(\text{ Impiegati JOIN Reparti })$			

Possiamo esprimere il join naturale usando il prodotto cartesiano? Per poterlo fare bisognerebbe cambiare il nome di Reparto in IdReparto (codice) in modo da ottenere tutte le coppie, e per ottenere il join naturale dovrei fare l'uguaglianza fra Reparto e IdReparto (codice).

Impiegati		Reparti	
Impiegato	Reparto	Reparto	Capo

### Impiegati NAT\_JOIN Reparti

$\text{PROJ}_{\text{Impiegato}, \text{Reparto}, \text{Capo}} (\text{SEL}_{\text{Reparto} = \text{Codice}} \\ (\text{Impiegati JOIN REN}_{\text{Codice} \leftarrow \text{Reparto}} (\text{Reparti}) ))$

## Esempi

Impiegati	Matricola	Nome	Età	Stipendio
	7309	Rossi	34	45
	5998	Bianchi	37	38
	9553	Neri	42	35
	5698	Bruni	43	42
	4076	Mori	45	50
	8123	Lupi	46	60

Supervisione	Impiegato	Capo
	7309	5698
	5998	5698
	9553	4076
	5698	4076
	4076	8123

Vi è un vincolo di integrità referenziale fra matricola e impiegato , la chiave primaria di impiegati è matricola, e la chiave primaria di supervisione è la combinazione fra impiegato e capo; quindi, ogni impiegato può avere più di un capo, e quindi il capo potrà apparire più volte.

➤ Trovare matricola, nome, età e stipendio degli impiegati che guadagnano più di 40.

Per far ciò ci basta utilizzare la **selezione**, specificando la condizione dello stipendio >40:

$\text{SEL}_{\text{Stipendio} > 40} (\text{Impiegati})$

Se invece l'esercizio ci avesse chiesto solo matricola, nome ed età degli impiegati che guadagnano più di 40 avremo dovuto fare la **proiezione** sul risultato della **selezione**:

$\text{PROJ}_{\text{Matricola, Nome, Età}} (\text{SEL}_{\text{Stipendio} > 40} (\text{Impiegati}))$

➤ Trovare i capi (matricola) degli impiegati che guadagnano più di 40:

Innanzitutto, si fa la **selezione** come prima con cui si estrapolano solo gli impiegati che guadagnano uno stipendio > 40, dopodiché si ricerca la corrispondenza fra il campo **impiegato** (tabella supervisione) e il campo **matricola** (tabella impiegati) con il **theta-join**, e infine si fa la **proiezione** scegliendo come unico campo i capi. Quindi otterremo una tabella con solamente i capi, i cui gli impiegati rispettano quella particolare condizione.

$\text{PROJ}_{\text{Capo}} (\text{Supervisione}$   
 $\text{JOIN}_{\text{Impiegato}=\text{Matricola}}$   
 $(\text{SEL}_{\text{Stipendio}>40}(\text{Impiegati})))$

Se avessimo voluto anche i nomi dei capi, non potrei fare la proiezione dei nomi perché otterrei i nomi degli impiegati che hanno lo stipendio > 40, ma dovremmo fare un altro **join** con condizione **capo = matricola**, e infine la **proiezione** su nome.

$\text{PROJ}_{\text{Nome}, \text{Stipendio}} ($   
 $\text{Impiegati JOIN}_{\text{Matricola}=\text{Capo}}$   
 $\text{PROJ}_{\text{Capo}} (\text{Supervisione}$   
 $\text{JOIN}_{\text{Impiegato}=\text{Matricola}} (\text{SEL}_{\text{Stipendio}>40}(\text{Impiegati}))))$

Nel caso di prima posso avere dei **capi duplicati**? No, perché l'**algebra relazionale** rimuove i **duplicati**.

$\text{PROJ}_{\text{Capo}} (\text{SEL}_{\text{Stipendio}>40} ($   
 $\text{Supervisione JOIN}_{\text{Impiegato}=\text{Matricola}} (\text{Impiegati})))$

Questa sintassi è equivalente a quella precedente.

Per rendere più leggibile le espressioni algebriche si utilizza **l'expression tree**, si parte dalla parte più innestata della nostra espressione. Nell'espressione precedente l'expression tree è così composto:

1. scriviamo la tabella in cui vogliamo operare;
2. successivamente la condizione che vogliamo eseguire, che nel nostro caso è la selezione con  $\text{Stipendio} > 40$ ;
3. la tabella ottenuta degli impiegati con stipendio > 40 viene messa in join con la tabella supervisione, verificando l'uguaglianza fra impiegato e matricola;
4. dal join risultante applichiamo la proiezione per ottenere solo i capi;
5. la tabella con i soli capi che viene messa in join con la tabella Impiegati, verificando la condizione  $\text{matricola} = \text{capo}$ ;
6. infine, si fa la proiezione sulla tabella risultante con nome e stipendio dei capi.

Nell'albero l'esecuzione parte dal basso verso l'alto e le operazioni sono collegate con degli **archi non orientati**.

### Expression tree



## ESERCIZIO

- ✓ Trovare gli impiegati che guadagnano più del proprio capo, mostrando matricola, nome e stipendio dell'impiegato e del capo, considerando come chiave primaria di supervisione solo impiegato.

Impiegato			Supervisione	
matricola	nome	stipendio	imp	capo
1	rossi	30	1	2
2	verdi	50	2	3
3	gialli	40		

1. Per prima cosa dobbiamo fare il **JOIN** fra Impiegati e Supervisione con **impiegato = matricola**, per trovare gli impiegati che hanno un capo;

matr	nome	stip	imp	capo
1	rossi	30	1	2
2	verdi	50	2	3

Non appare gialli perché lui non ha nessun capo.

2. Successivamente dobbiamo fare un altro **join** tra Impiegati e Supervisione in cui il **capo è uguale alla matricola**, per trovare i capi. Dato che il risultato lo dobbiamo mettere in **join** con il risultato precedente, dobbiamo **rinominare** i campi della tabella Impiegati che rappresentano solo i capi, in caso non lo si facesse avremo dei campi con il nome uguale. Quindi rinomino la tabella con il nome Capi, e ne rinomino anche i nomi degli attributi.

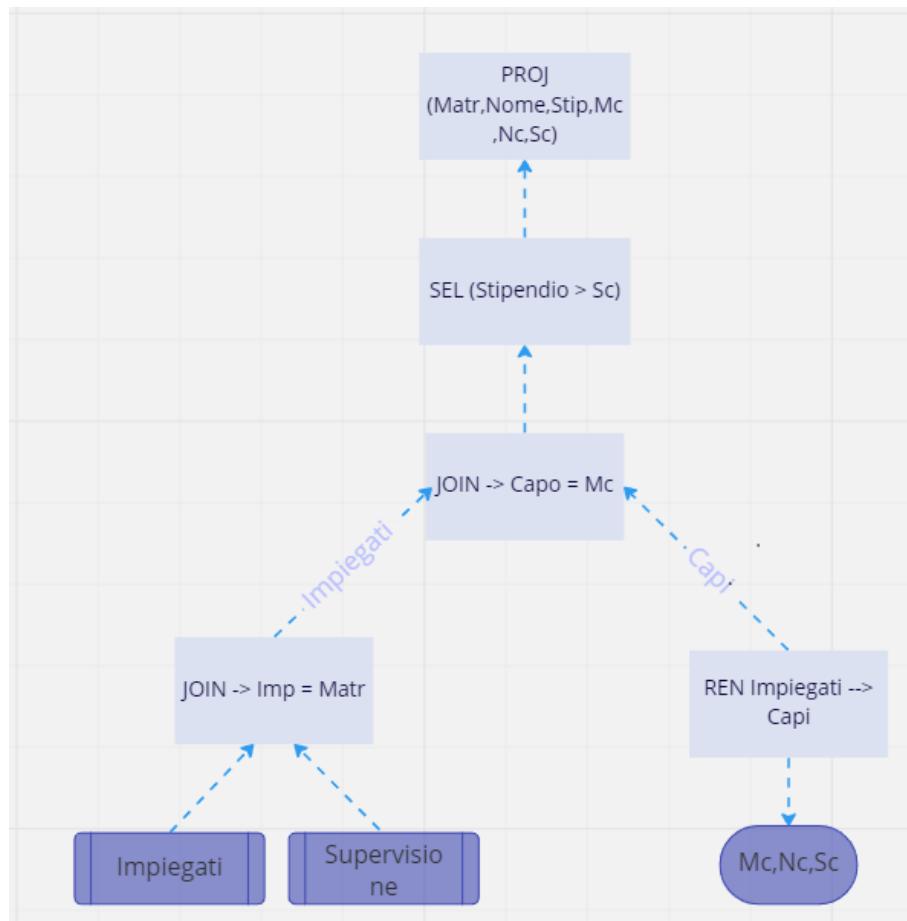
matr	nome	stip	imp	capo	mc	nc	sc
1	rossi	30	1	2	2	verdi	50
2	verdi	50	2	3	3	gialli	40

3. Ottengo quindi una tabella in cui ho i valori degli Impiegati e i valori dei Capi rinominati; quindi, ci basta effettuare la **selezione** verificando la condizione **Stipendio > Sc**.

matr	nome	stip	imp	capo	mc	nc	sc
2	verdi	50	2	3	3	gialli	40

4. Infine, proiettiamo i risultati rimuovendo le colonne **imp** e **capo**, in modo da visualizzare solamente i dati degli impiegati il cui lo stipendio è > di quello del capo.

**PROJ**<sub>Matr,Nome,Stip,MatrC,NomeC,StipC</sub>  
**(SEL**<sub>Stipendio>StipC</sub>**)**  
**REN**<sub>MatrC,NomeC,StipC,EtàC ←</sub>  
**Matr,Nome,Stip,Età****(Impiegati)**  
**JOIN**<sub>MatrC=Capo</sub>  
**(Supervisione JOIN**<sub>Impiegato=Matricola</sub>  
**Impiegati))**



☞ Trovare le matricole dei capi i cui impiegati guadagnano tutti più di 40

$\text{PROJ}_{\text{Capo}}(\text{Supervisione}) - \text{PROJ}_{\text{Capo}}(\text{Supervisione} \text{ JOIN } \text{Impiegato}=\text{Matricola} \text{ (SEL}_{\text{Stipendio} \leq 40}(\text{Impiegati}))$

Selezioniamo solo gli impiegati che hanno lo stipendio  $\leq 40$  e il risultato lo mettiamo in **join** con la tabella Supervisione con i soli capi (utilizzando la [proiezione](#)) considerando come condizione **impiegato = matricola**. Proiettiamo i capi dalla tabella Supervisione e rimuoviamo ([sottrazione](#)) i capi che abbiamo ottenuto prima con il precedente join, così da ottenere tutti i capi che hanno impiegati che guadagnano più di 40.

Se avessimo fatto prima il **join** fra Impiegato e Supervisione con **impiegato = matricola**, e successivamente avessimo selezionato coloro con lo stipendio  $\leq 40$ , e infine avessimo fatto la proiezione dei capi, avremo ottenuto solo i capi che hanno almeno un impiegato con uno stipendio  $> 40$ .

A volte può capitare che un attributo di una selezione **non abbia nessun valore**, cioè che abbia il valore **NULL**. In questo caso la riga contenente quel valore **non viene restituita dalla selezione** in quanto la condizione atomica è vera solo per valori non nulli.

### Impiegati

Matricola	Cognome	Filiale	Età
7309	Rossi	Roma	32
5998	Neri	Milano	45
9553	Bruni	Milano	NULL

$\text{SEL}_{\text{Età} > 40}(\text{Impiegati})$

In questo caso quindi la riga contenente Bruni non verrà restituito dalla selezione.

$$\text{SEL}_{\text{Età}>30}(\text{Persone}) \cup \text{SEL}_{\text{Età}\leq 30}(\text{Persone}) \neq \text{Persone}$$

In questo esempio possiamo vedere che la Selezione con condizione  $\text{età} > 30$  unito alla selezione con condizione  $\text{età} \leq 30$  è diversa dalla tabella completa Persone, questo è dovuto al fatto che le selezioni vengono valutate separatamente; quindi, prima si fa l'una e successivamente si applica la seconda sul risultato della prima.

$$\text{SEL}_{\text{Età}>30 \vee \text{Età}\leq 30}(\text{Persone}) \neq \text{Persone}$$

Anche quando si tratta di avere due condizioni atomiche, quest'ultime vengono valutate separatamente.

Per porre delle condizioni sugli attributi che possono avere come valore NULL, si utilizzano le condizione **IS NULL** e **IS NOT NULL**. Quindi se io volessi estendere l'unione precedente anche con valori NULL dovremmo fare così:

$$\begin{aligned} \text{SEL}_{\text{Età}>30}(\text{Persone}) \cup \text{SEL}_{\text{Età}\leq 30}(\text{Persone}) \cup \\ \text{SEL}_{\text{Età IS NULL}}(\text{Persone}) \\ = \\ \text{SEL}_{\text{Età}>30 \vee \text{Età}\leq 30 \vee \text{Età IS NULL}}(\text{Persone}) \\ = \\ \text{Persone} \end{aligned}$$

Come abbiamo visto precedentemente le **viste** sono delle rappresentazioni diverse per gli stessi dati, e sono definite a partire dalle tabelle vere e proprie, le quali possono essere:

- **Derivate**: cioè le relazioni il cui contenuto è funzione del contenuto di altre relazioni (definito per mezzo di interrogazioni), e possono essere definite su altre relazioni derivate;
- **Di base**: il cui contenuto è autonomo.

## Viste, esempio

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

In questo esempio una vista potrebbe mettermi assieme le informazioni di un impiegato e del suo capo, e questo lo si può fare con il natural Join:

$$\begin{aligned} \text{Supervisione} = \\ \text{PROJ}_{\text{Impiegato}, \text{Capo}}(\text{Afferenza NAT_JOIN Direzione}) \end{aligned}$$

A questa nuova vista possiamo dargli un nome, che in questo caso è Supervisione; quindi, è possibile interrogare le viste sostituendo alla vista la sua definizione:

$\text{SEL}_{\text{Capo}=\text{'Leoni'}}$  (Supervisione)  
 viene eseguita come  
 $\text{SEL}_{\text{Capo}=\text{'Leoni'}} \cdot$   
 $\text{PROJ}_{\text{Impiegato}, \text{Capo}} (\text{Afferenza JOIN Direzione})$

Le motivazioni per utilizzare le viste sono le seguenti:

- Corrisponde allo schema esterno, quindi ogni utente può vedere ciò che gli interessa e solamente ciò che è autorizzato a vedere;
- È uno strumento di programmazione in quanto può semplificare la scrittura di interrogazioni;
- Permette l'utilizzo di programmi esistenti su schemi ristrutturati.

L'algebra relazionale ha alcuni limiti:

- Ci sono interrogazioni interessanti non esprimibili:
  - ↗ Calcolo di valori derivati: possiamo solo estrarre valori, non calcolarne di nuovi, sia a livello di tupla o di singolo valore (conversioni, somme, etc.), che su insiemi di tuple (somme, medie, etc.);
  - ↗ Interrogazioni inerentemente ricorsive, come la chiusura transitiva.

Un esempio di **chiusura transitiva** è: per ogni impiegato, trovare tutti i superiori (cioè il capo, il capo del capo, e così via), non esiste però in algebra relazionale la possibilità di esprimere questa interrogazione.

## Esercizi

Consideriamo questo schema:

- MAGAZZINO (CodiceM, Città, Via, NumeroCivico) ;
- AZIENDA (CodiceA, NomeAzienda, Città, CAP, Telefono);
- PRODOTTO (CodiceP, NomeProdotto, Azienda, Prezzo)  
FK: Azienda = AZIENDA (CodiceA);
- STOCCAGGIO (Prodotto, Magazzino, Quantità)  
FK: Prodotto= PRODOTTO (CodiceP);  
Magazzino= MAGAZZINO(CodiceM).

In stoccaggio la doppia chiave esterna ci dice che posso avere lo stesso prodotto stoccati in magazzini diversi, allo stesso modo vale l'inverso; quindi, anche lo stesso magazzino può stoccare più prodotti diversi.

- ↗ Elencare il nome di tutti i prodotti che costano meno di 10 euro:

**$\text{PROJ}_{\text{Nome}}(\text{SEL}_{\text{Prezzo} < 10}(\text{PRODOTTO}))$**



- Elencare i numeri civici dei magazzini di via Ripamonti a Milano:



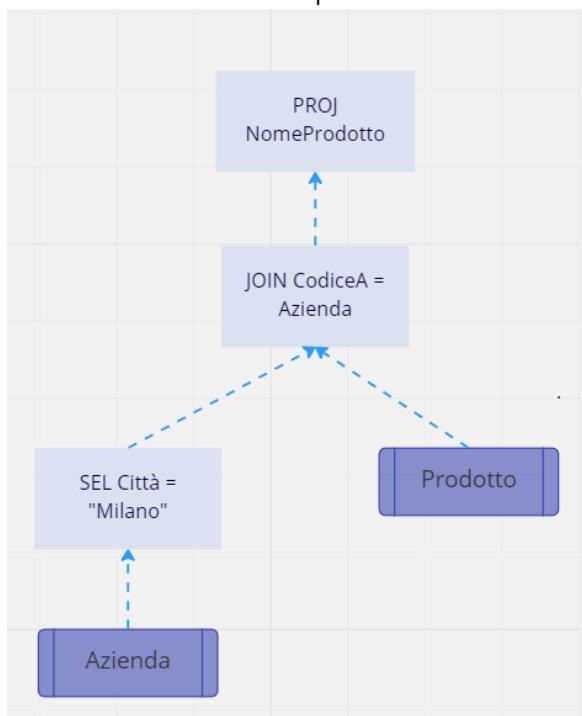
**PROJ<sub>NumeroCivico</sub>(  
SEL<sub>Città="Milano" AND Via="Ripamonti"</sub>(MAGAZZINO))**

- Elencare il codice dei prodotti stoccati in un magazzino avente lo stesso codice del prodotto (può capitare che per caso il codice del prodotto sia uguale a quello del magazzino, noi stiamo cercando questo caso):



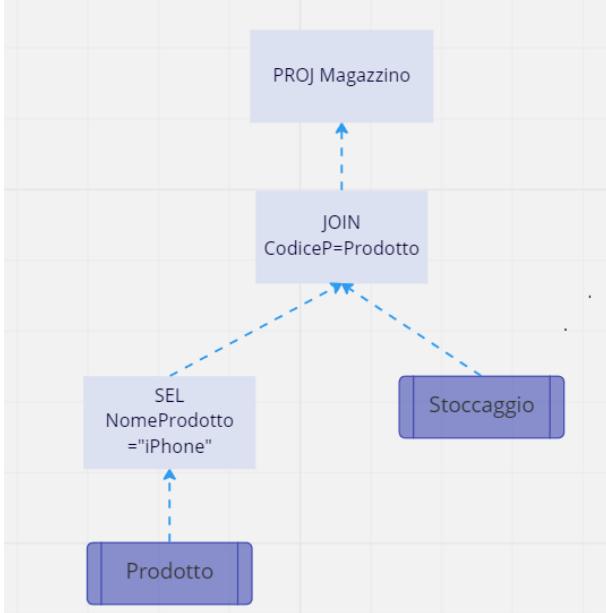
**PROJ<sub>Prodotto</sub>(SEL<sub>Prodotto=Magazzino</sub>(STOCCAGGIO))**

- Elencare il nome dei prodotti delle aziende di Milano:



**PROJ<sub>NomeProdotto</sub>(PRODOTTO  
JOIN<sub>CodiceA = Azienda</sub>(SEL<sub>Città = 'Milano'</sub>(AZIENDA)))**

- Elencare il codice di tutti i magazzini in cui è stoccatto un prodotto di nome "iPhone":



**PROJ<sub>Magazzino</sub> (STOCCAGGIO)**

**JOIN<sub>Prodotto = CodiceP</sub> (**

**SEL<sub>NomeProdotto = 'iPhone'</sub>(PRODOTTO)))**

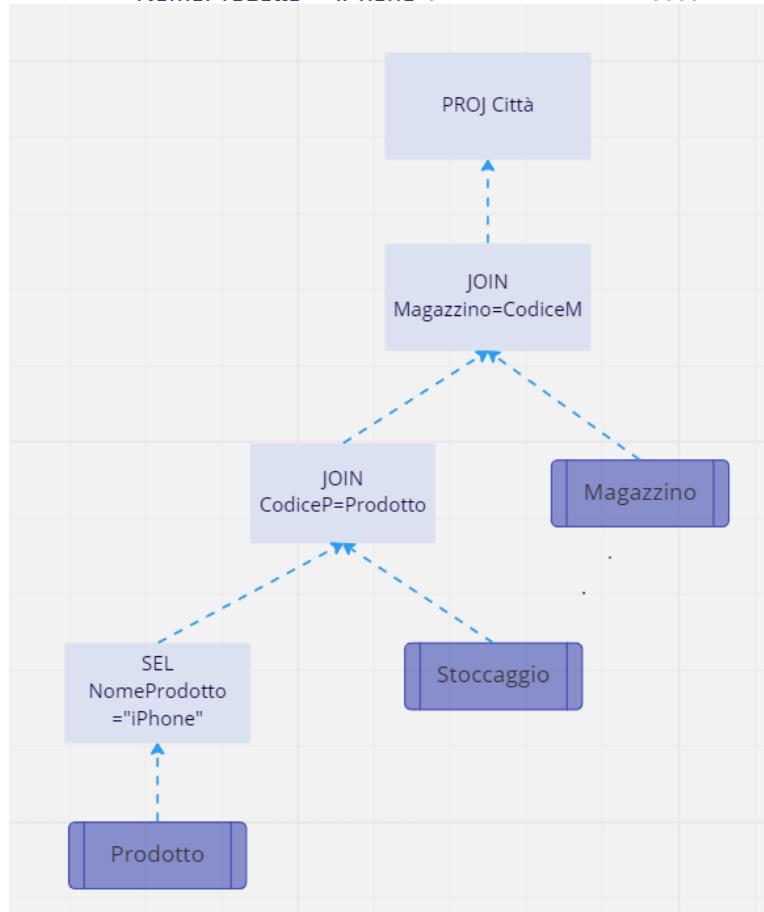
- Elencare la città di tutti i magazzini in cui è stoccatto un prodotto di nome "iPhone":

**PROJ<sub>Città</sub> (MAGAZZINO)**

**JOIN<sub>Magazzino=CodiceM</sub> (STOCCAGGIO**

**JOIN<sub>Prodotto = CodiceP</sub> (**

**SEL<sub>NomeProdotto = 'iPhone'</sub>(PRODOTTO))))**

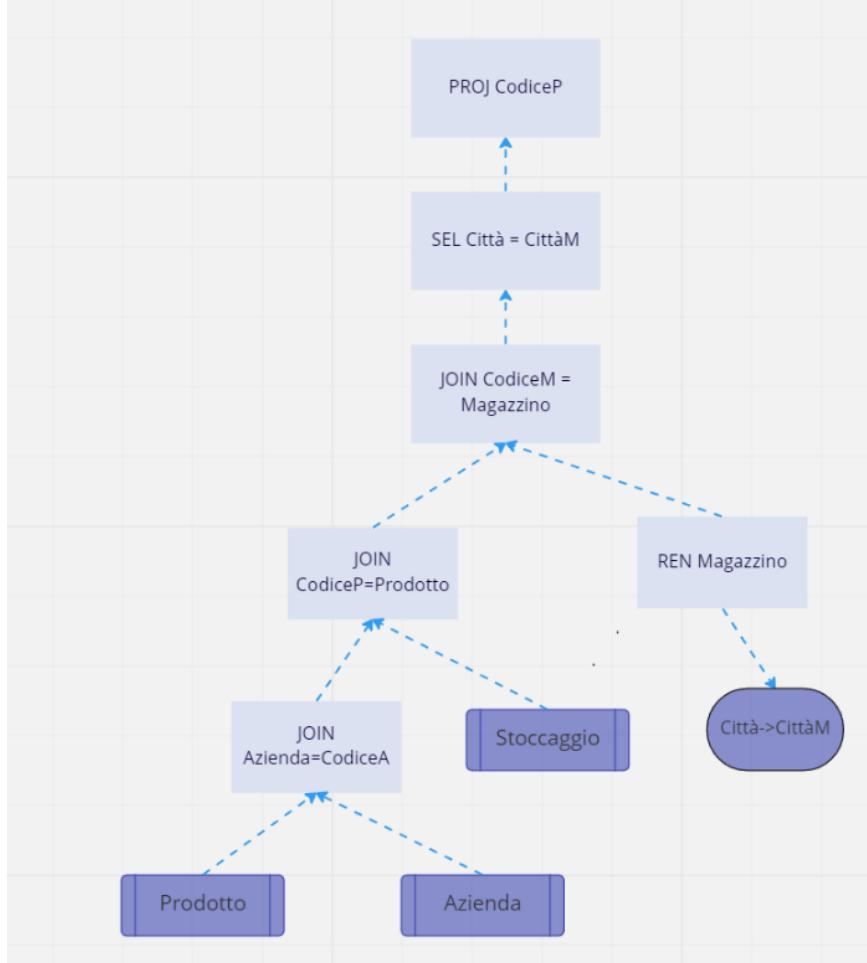


✓ Elencare il codice di quei prodotti che sono stoccati nella città dell'azienda che li produce:

1. Per prima cosa troviamo la corrispondenza fra i prodotti e l'azienda che li produce, facendo il **JOIN** con la condizione **Azienda** (chiave esterna in **Prodotto**) = **CodiceA** (chiave primaria in **Azienda**). Così otteniamo una tabella contenente i dati delle tabelle **Prodotto** e **Azienda**;
2. Poi troviamo la corrispondenza fra i prodotti e dove quest'ultimi sono stati stoccati; quindi, facciamo il **JOIN** fra il risultato del JOIN precedente e la tabella **Stoccaggio** verificando la condizione **CodiceP = Prodotto**. Così otteniamo una tabella in cui abbiamo i dati delle tabelle **Prodotto**, **Azienda** e **Stoccaggio**;
3. Successivamente dobbiamo trovare la corrispondenza fra **CodiceM** (Magazzino) e Magazzino (presente nei JOIN precedenti), quindi facciamo il **JOIN** verificando quest'uguaglianza. Quest'operazione però porta ad avere due campi uguali, cioè le città delle Aziende e le città dei Magazzini, per evitare questo facciamo una **ridenominazione** prima del JOIN con cui cambiamo il nome della città del magazzino in **CittàM**;
4. Facciamo la **selezione** con condizione **Città=CittàM**;
5. Infine, **proiettiamo** i codici dei prodotti.

```

PROJ CodiceP(
    SEL Città=CittàM(
        REN CittàM ← Città(MAGAZZINO)
        JOIN Magazzino=CodiceM (
            STOCCAGGIO JOIN Prodotto=CodiceP (
                PRODOTTO JOIN Azienda=CodiceA (AZIENDA)
            )
        )
    )
)
)
)
)
)
```



- Elencare il codice, il nome e il prezzo dei prodotti che costano più di 100:

**PROJ**<sub>CodiceP, NomeProdotto, Prezzo</sub>(  
**(SEL**<sub>Prezzo > 100</sub> (**PRODOTTO**))

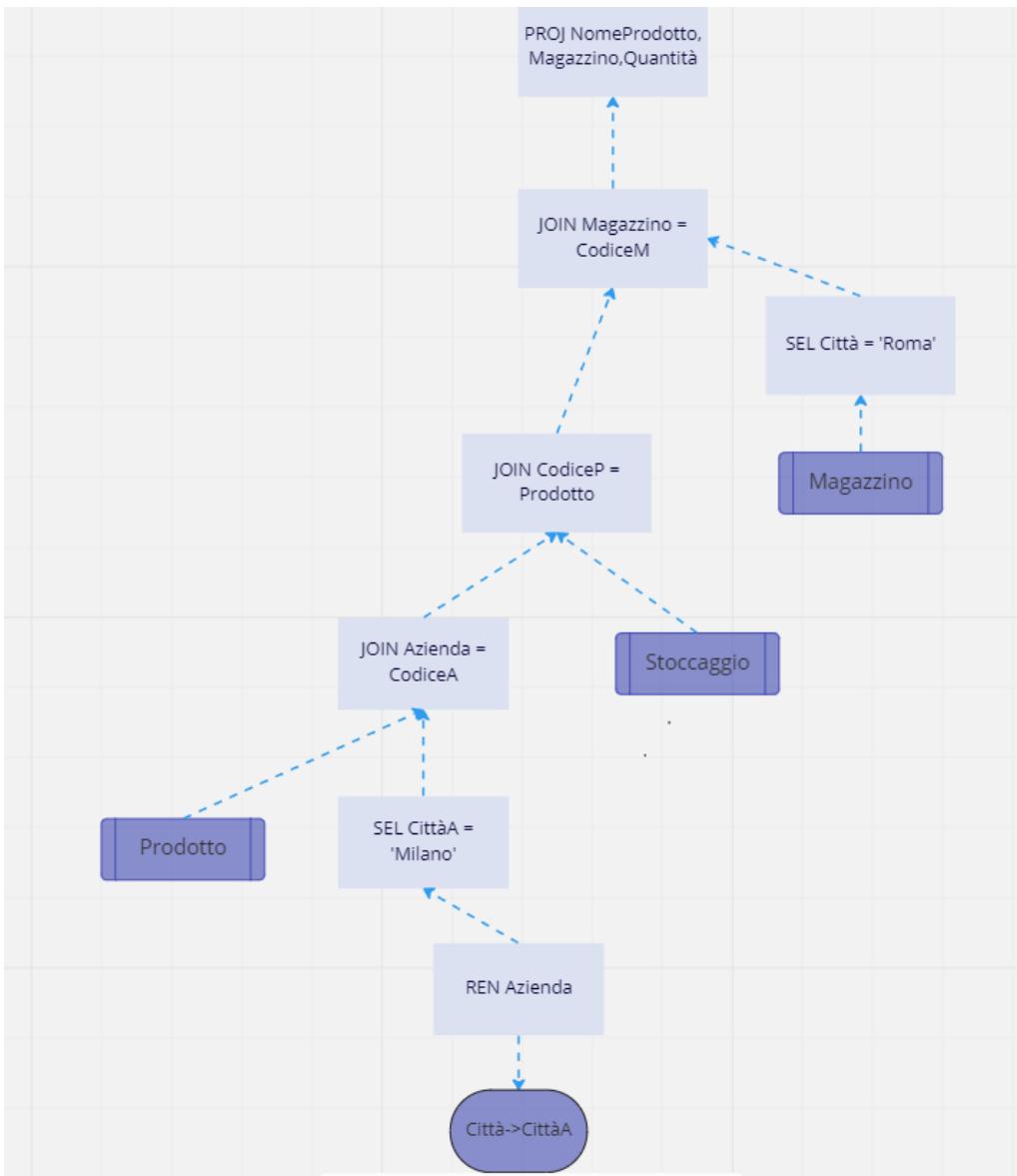
- Elencare il nome e il prezzo dei prodotti dell'azienda di nome "Apple":

**PROJ**<sub>NomeProdotto, Prezzo</sub>(**PRODOTTO**  
**JOIN**<sub>Azienda = CodiceA</sub>  
**(SEL**<sub>NomeAzienda = "Apple"</sub> (**AZIENDA**)))

- Selezionare i prodotti di aziende milanesi stoccati in magazzini di Roma, indicando per ogni stoccaggio il nome del prodotto, il codice del magazzino, e la quantità stoccatata:

1. Per prima cosa **rinominiamo** il campo città di Azienda in CittàA, questo ci servirà per dopo;
2. Poi **selezioniamo** dalla Azienda solo i record in cui vi è la corrispondenza CittàA = 'Milano', in modo tale da trovare solo le aziende milanesi;
3. Successivamente dobbiamo trovare la corrispondenza fra Azienda (Prodotto) e CodiceA (Azienda), quindi facciamo il **JOIN** verificando quest'uguaglianza, quest'operazione ci restituirà una tabella contenente i valori delle tabelle di **Azienda** e **Prodotto** in cui vi è la corrispondenza fra la chiave primaria di Azienda e la chiave esterna di Azienda in Prodotto;
4. Facciamo un ulteriore **JOIN** in cui verifichiamo la corrispondenza fra Prodotto (Stoccaggio) e CodiceP (ottenuto dal JOIN precedente). In questo modo otteniamo una tabella contenente i dati delle aziende milanesi, i prodotti, che sono stati prodotti in quest'ultime e i dati di stoccaggio di questi prodotti.
5. Facciamo un ultimo **JOIN** con cui troviamo la corrispondenza fra CodiceM (chiave primaria di Magazzino) e Magazzino (presente dai JOIN precedenti), così da ottenere una tabella contenente i dati delle aziende milanesi, dei prodotti, dello stoccaggio, e dei relativi magazzini;
6. Quindi **selezioniamo** solo quei campi che hanno come Città = 'Roma', dove Città è la città dei Magazzini, e non vi è confusione, perché per prima cosa avevamo cambiato il nome delle città delle Aziende;
7. Infine, **proiettiamo** il NomeProdotto, il Magazzino e la Quantità.

**PROJ**<sub>NomeProdotto, Magazzino, Quantità</sub>(  
**(SEL**<sub>Città = "Roma"</sub> (**MAGAZZINO**))**JOIN**<sub>Magazzino = CodiceM</sub>  
**STOCCAGGIO JOIN**<sub>Prodotto = CodiceP</sub> (**PRODOTTO**  
**JOIN**<sub>Azienda = CodiceA</sub> (**SEL**<sub>CittàA = "Milano"</sub>(  
**REN**<sub>CittàA ← Città</sub> (**AZIENDA**))))



- Elencare il codice di tutti i magazzini in cui non è stoccatato nessun prodotto:



**REN<sub>Magazzino</sub> ← CodiceM (PROJ<sub>CodiceM</sub>(MAGAZZINO)) – PROJ<sub>Magazzino</sub> (SEL<sub>Quantità > 0</sub> (STOCCAGGIO))**

- ✓ Elencare le città che sono sede di una azienda o di un magazzino (o di entrambi):

**PROJ<sub>Città</sub>(MAGAZZINO) ∪ PROJ<sub>Città</sub>(AZIENDA)**

Se avessimo voluto sapere anche le città che non sono sede contemporaneamente sia di un magazzino che di un'azienda avremmo dovuto agire come sopra e sottrarre l'intersezione fra i due.

- ✓ Elencare le coppie di prodotti diversi che hanno lo stesso nome (indicando il codice di ognuno): Questo è il risultato che vorremo ottenere:

Codice	Nome	P1	P2
P1	N1	P1	P3
P2	N1	P2	P3
P3	N1		
P4	N2		

Per ottenerlo bisogna applicare il **prodotto cartesiano**, così che prendo tutte le coppie di prodotti.

**PROD1 = REN<sub>C1, N1, A1, P1</sub> (PRODOTTO)**

**PROD2 = REN<sub>C2, N2, A2, P2</sub> (PRODOTTO)**

**PROJ<sub>C1,C2</sub> (SEL<sub>(N1=N2) ∧ (C1≠C2)</sub> (PROD1 × PROD2))**

Con C1 ≠ C2 ottengo anche coppie ripetute: ad es. [33, 66] e [66, 33], per tenere una sola coppia, invece di ≠ uso > :

**PROJ<sub>C1,C2</sub> SEL<sub>(N1=N2) ∧ (C1>C2)</sub> (PROD1 × PROD2 ))**

Così ottengo solo [66,33].

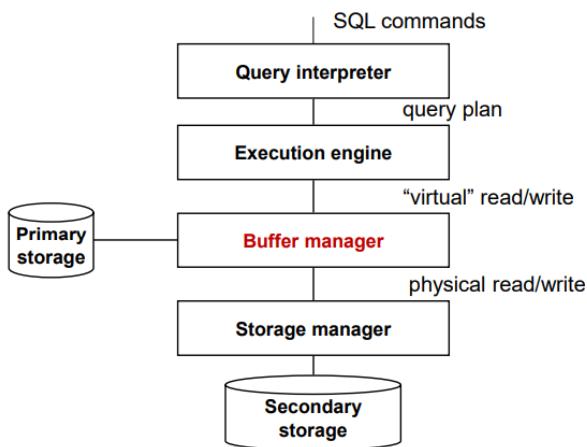
- ✓ Elencare il codice dei prodotti stoccati in almeno 2 magazzini diversi:

**STOC1 = REN<sub>P1, M1, Q1</sub> (STOCCAGGIO)**

**STOC2 = REN<sub>P2, M2, Q2</sub> (STOCCAGGIO)**

**PROJ<sub>P1</sub> (SEL<sub>(P1=P2) ∧ (M1≠M2)</sub> (STOC1 × STOC2))**

# Organizzazione fisica e gestione delle interrogazioni



memoria principale) che prende i dati dalla memoria secondaria.

I programmi possono fare riferimento solo a dati in memoria principale, le basi di dati devono essere scritte in memoria secondaria per due motivi:

- Dimensioni;
- Persistenza.

I dati in memoria secondaria possono essere utilizzati solo se prima trasferiti in memoria principale.

I dispositivi di memoria secondaria sono organizzati in **blocchi** di lunghezza fissa (alcuni KB), le uniche operazioni sui dispositivi sono la **lettura e la scrittura di un intero blocco**.

Una **pagina** è un' unità di suddivisione logica della RAM, per semplicità assumiamo → dimensione pagina = dimensione blocco.

Una differenza principale fra HardDisk e disco riguarda la **velocità di acquisizione dei dati**. Nel HardDisk tradizionale (HDD) il tempo di posizionamento della testina, tempo di latenza, tempo di trasferimento è in media di **1 - 10 ms**. Invece una **memoria secondaria** con unità a stato solido (SSD) è più costosa ma ha dei tempi di accesso circa **50 volte minori** rispetto a HDD. Infine, troviamo anche l'accesso a **memoria terziaria** (tapes, optical jukebox) che ha un tempo in media di **5 - 60** secondi.

Il costo di un accesso a **memoria secondaria** è quattro o più ordini di grandezza maggiore di quello per operazioni in **memoria centrale**. Quindi il costo di una operazione del DBMS **dipende esclusivamente dal numero di accessi a memoria secondaria** (numero di blocchi che devono essere letti), gli accessi a blocchi "vicini" costano meno per via della **contiguità**.

In generale il **buffer** è un' area di memoria centrale, gestita dal DBMS (preallocata) e condivisa fra le transazioni, lo possiamo vedere come un "deposito temporaneo" delle porzioni di DB che il sistema sta utilizzando in un dato momento. Il suo scopo è quello di **ridurre il numero di accessi alla memoria secondaria**.

Il buffer manager si occupa principalmente di due aspetti:

- Del **contenuto** del buffer;
- Svolge la funzione di **direttorio** che per ogni pagina mantiene:
  - il file e il numero del blocco corrispondente in HardDisk;
  - due variabili di stato:
    - un **contatore** che indica quanti programmi utilizzano la pagina;

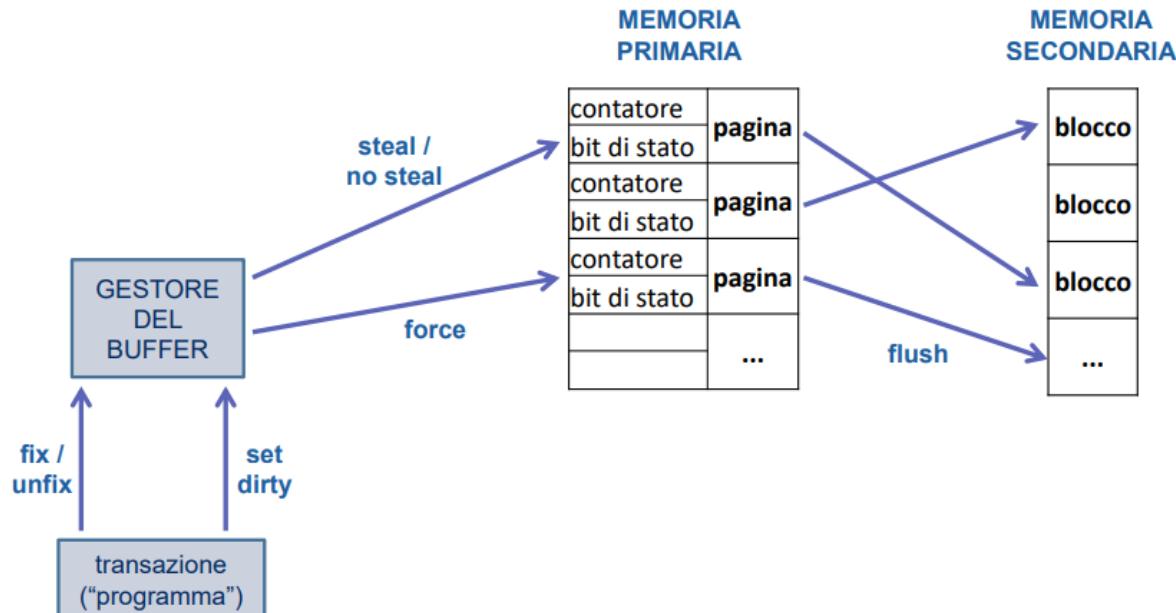
A livello più alto troviamo i comandi che vengono forniti dall'utente del DBMS, che vengono successivamente interpretati dal **Query interpreter** che cerca di fare il parsing del SQL; successivamente vi è il **query plan** che è un algoritmo che determina come eseguire la query vera e propria nel modo più efficiente possibile.

Dopo che si è deciso come effettuare la query, si passa all'**execution engine** che da una serie di comandi al buffer manager, comandi di tipo lettura e scrittura. Infine, troviamo lo **storage manager** che si occupa di gestire il buffer (che si trova nella memoria principale) e li trasferisce nella memoria secondaria.

- un **bit di stato** che indica se la pagina è "sporca", cioè se è stata modificata (la scrittura su memoria secondaria può essere differita).

Le sue **funzioni** sono:

- Riceve richieste di lettura e scrittura di blocchi (che contengono record);
- Utilizza il buffer quando possibile;
- Esegue le richieste accedendo alla memoria secondaria solo quando indispensabile;
- Esegue le primitive: fix, unfix, setDirty, force.



Ogni transazione farà delle richieste che possono essere del tipo **fix** o **unfix**, oppure aviserà il gestore del buffer tramite una **setDirty** che una pagina è stata modificata. Dopo una fix il buffer manager incrementerà il **contatore** di una determinata pagina, al contrario con una unfix lo decrementerà; invece, con la **setDirty** si modificherà il bit di stato. La **force** e la **flush** riguardano come i dati vengono trasferiti dalla RAM al disco.

Ricapitolando:

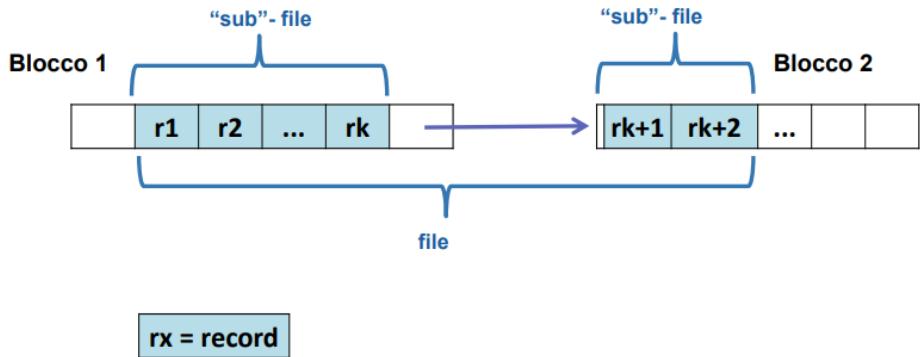
- **fix**: richiesta di una pagina → richiede una lettura in memoria secondaria solo se la pagina non è nel buffer e incrementa il contatore associato alla pagina;
- **setDirty**: comunica al buffer manager che la pagina è stata modificata;
- **unfix**: indica che la transazione ha concluso l'utilizzo della pagina → decremente il contatore associato alla pagina;
- **force**: trasferisce in modo sincrono una pagina in memoria secondaria (disco);
- **flush**: scrittura asincrona in memoria;

La fix funziona in questo modo: cerca la pagina nel buffer, se c'è, restituisce l'indirizzo, altrimenti, cerca una pagina libera nel buffer (avente contatore a zero) e la sovrascrive. Se ce n'è più di una? Si utilizza la politica FIFO, oppure Least Recently Used, o un'altra politica, e la pagina sovrascritta viene prima salvata su disco se necessario.

Se non ci dovessero essere dei contatori a zero si hanno due alternative:

- **steal**: selezione di una "vittima" (pagina occupata del buffer);
- **no-steal**: l'operazione viene posta in attesa.

L'**organizzazione interna** dei file è gestita direttamente dal DBMS, sia la struttura all'interno dei singoli blocchi, sia la distribuzione dei record nei blocchi.



I dati di una tabella vengono inseriti in diversi **blocchi**, che sono costituiti da diversi record. L'intera tabella rappresenta un **file** (costituita quindi da più blocchi).

Si hanno due alternative per scrivere i record nei blocchi:

- Formato dei record fisso oppure auto-descrittivo;
- Lunghezza dei campi fissa oppure variabile.

## Formato fisso e lunghezza fissa

### Employee record

- (1) E#, 2 byte integer
- (2) E.name, 10 char.
- (3) Dept, 2 byte code

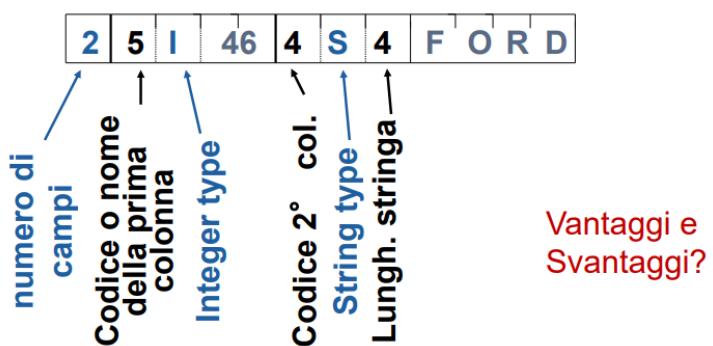
55	s m i t h	02
83	j o n e s	01

Schema (nel  
direttorio)

Record  
(in altri file)

La cosa più semplice è avere un **formato fisso**, quindi non si ha nessuna flessibilità nello schema, e una lunghezza fissa del record. Lo svantaggio principale può essere che, come si vede nell'esempio i nomi, sono più piccoli rispetto alla lunghezza stabilita e questo porta ad uno **spreco di memoria**.

## Formato auto-descrittivo e lunghezza variabile



In questo caso si ha la massima libertà nel memorizzare i record, per farlo si inseriscono i metadati all'interno del record (come, ad esempio, il numero di campi).

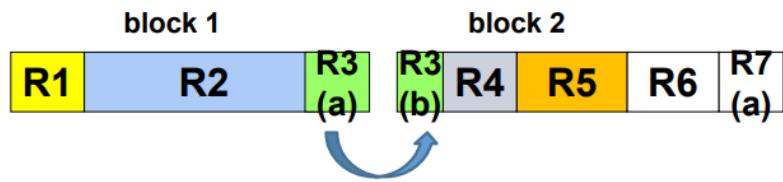
Nel caso in cui avessimo dei **dati eterogeni** fra di loro si preferisce utilizzare questo sistema, se invece i dati sono molto simili si preferisce utilizzare quello a formato fisso. Quindi la scelta dipende da come sono i dati.

Dopo aver capito come costituire internamente un record, il problema successivo è come scrivere i record all'interno dei blocchi. La metodologia può essere:

- **Unspanned**: si ha esattamente un record per blocco (si può presentare uno spreco di spazio):



- **Spanned**: più record per blocco:



Lo svantaggio principale di questo metodo è il costo di lettura e scrittura che risulta molto più lungo (l'accesso al blocco), in quanto se si dovesse leggere R3 bisogna leggere sia dal blocco 1 che dal blocco 2. Lo spanned è necessario per record più grandi di un blocco.

Quando si vanno a scrivere i record un'altra caratteristica è come scriverli, se in ordine di riga ([row store](#)) o in ordine di colonna ([column store](#)):

## Row Store

- Esempio: Relazione “Acquisto”
- Acquisto(id, cust, prod, store, price, date, qty)

BLOCK1						
id1	cust1	prod1	store1	price1	date1	qty1
BLOCK2						
id2	cust1	prod2	store2	price2	date2	qty2
BLOCK3						
id3	cust3	prod1	store1	price1	date3	qty3

## Column Store

- Esempio: Relazione “Acquisto”
- Acquisto(id, cust, prod, store, price, date, qty)

BLOCK1		BLOCK2		BLOCK3	
id1	cust1	id1	prod1	id1	price1
id2	cust1	id2	prod2	id2	price2
id3	cust3	id3	prod1	id3	price1
id4	cust4	id4	prod1	id4	price4
...	...	...	...	...	...

Gli ID possono essere scritti esplicitamente oppure essere impliciti in base all'ordine

## Column Store - variante

- Esempio: Relazione “Acquisto”
- Acquisto(id, cust, prod, store, price, date, qty)

BLOCK1		BLOCK2		BLOCK3	
id1;id2	cust1	id1;id3;id4	prod1	id1;id3	price1
id3	cust3	id2	prod2	id2	price2
id4	cust4	...	...	id4	price4
...	...	...	...	...	...
...	...	...	...	...	...

Gli ID devono essere espliciti

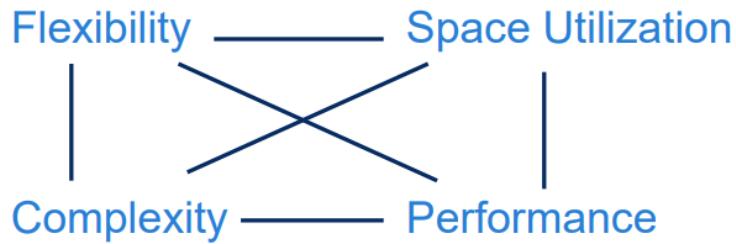
Un'ottimizzazione per risparmiare spazio è quella di mettere assieme i campi che hanno lo stesso id, come in questo esempio.

Il vantaggio del [Column Store](#) è la lettura efficiente per applicazioni di “data mining” e statistiche; invece, il vantaggio del [Row Store](#) è che è più efficiente per letture e scritture di molteplici campi dello stesso record.

Esiste un [ordinamento](#) fra le tuple che può essere rilevante ai fini della gestione:

- Organizzazione [disordinata](#): si ha un ordinamento fisico dei record ma non logico, gli inserimenti vengono effettuati al posto di record cancellati, oppure in coda (con riorganizzazioni periodiche);
- Organizzazione [ordinata](#): l'ordinamento dei record è coerente con quello di un campo, viene usata principalmente per memorizzare gli indici.

Ci sono molti modi per organizzare i record in memoria. Qual è il migliore? Bisogna considerare diversi aspetti:



## Livello fisico del DBMS: Indici

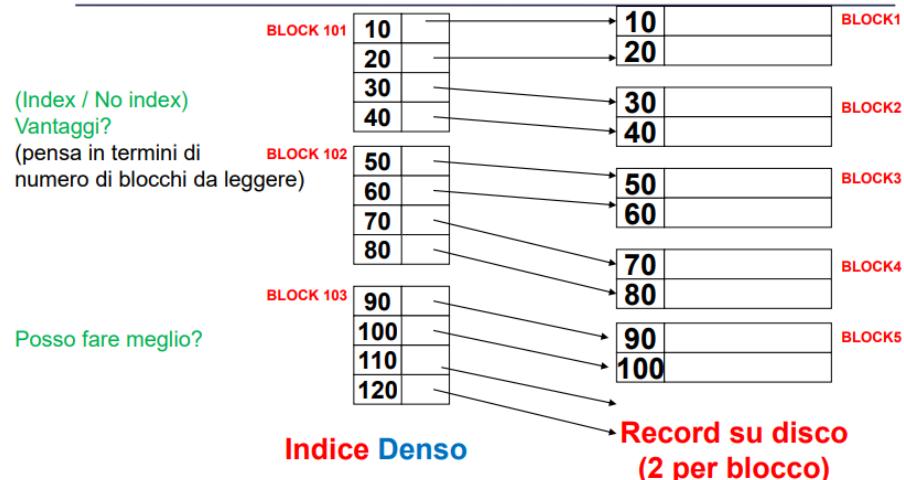
Un indice è struttura ausiliaria per l'**accesso** (efficiente) ai record, il quale può essere sulla base dei valori di uno o più campi. I campi sono detti **chiave** (ma non sono necessariamente identificanti).

Un indice in un DBMS lo possiamo pensare come un **indice analitico di un libro**, cioè una lista di coppie (termine, pagina), ordinata alfabeticamente sui termini, posta in fondo al libro e separabile da esso.

Si hanno diversi tipi di indici:

- Indice **primario**: viene applicato su un campo sul cui ordinamento è basata la memorizzazione. Ad esempio, se applicassimo un indice su matricola, e la tabella fosse ordinata per matricola allora questo è un indice primario;
- Indice **secondario**: viene applicato su una struttura non ordinata, o su una struttura ordinata, ma su un campo che non determina l'ordinamento;
- Indice **denso**: contiene un'entrata per ciascun valore del campo chiave;
- Indice **sparso**: contiene un numero di entrate inferiore rispetto al numero di valori diversi del campo chiave.

### Indice denso vs sparso      Scrittura ordinata



Abbiamo a destra una struttura ordinata, in cui i blocchi hanno solo due record, di cui il primo è la chiave.

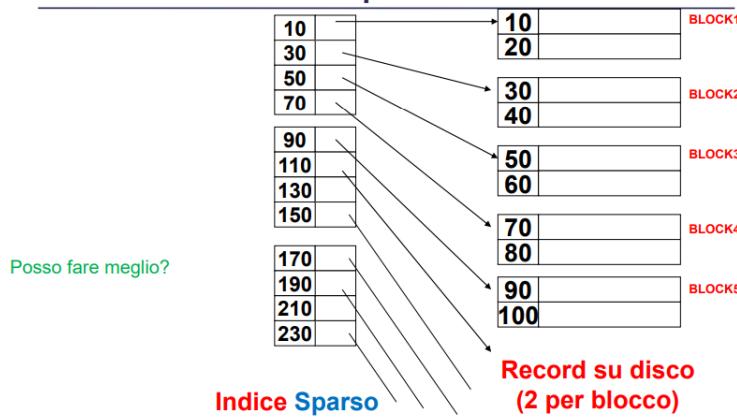
In questo caso con un **indice denso** abbiamo un valore per ogni valore della chiave. Quindi l'indice è una coppia contenente la chiave e il puntatore al record.

Se volessimo cercare la chiave 40, facendo una semplice scansione, bisognerebbe fare due accessi, uno al block1, dove

non si trova la chiave cercata e uno al block2.

Con l'utilizzo dell'indice denso faremo comunque 2 accessi, uno al block 101 e uno al block2.

## Indice denso vs sparso



Per migliorare l'efficienza potremmo mettere nell'indice un unico valore che contiene il puntatore al blocco in cui sono contenuti in realtà due record.

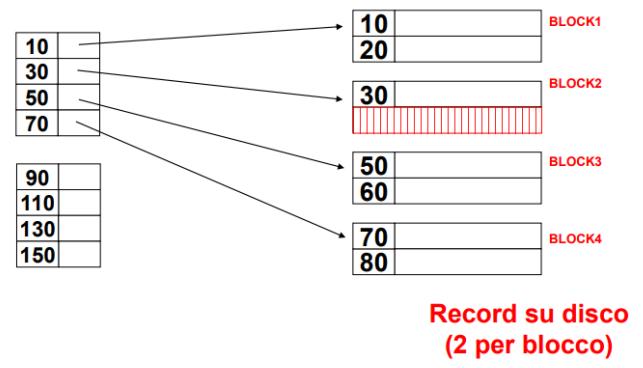
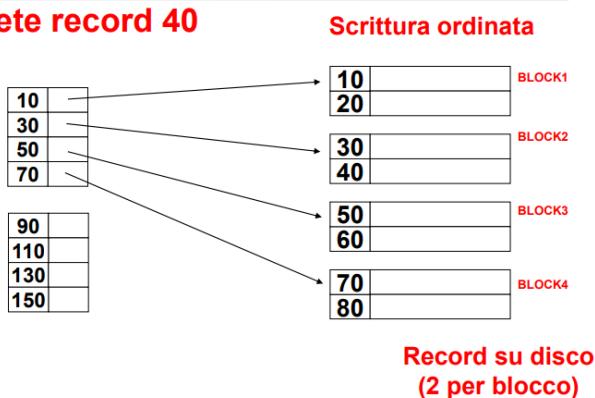
Quindi in questo caso non stiamo utilizzando più l'indice denso ma **l'indice sparso**.

Se dovessi cercare la chiave 60 nell'indice non la si trova, ma si va a verificare nel blocco corrispondente all'indice 50, che si trova prima del valore 60. Grazie a questo si fanno due accessi rispetto a 3 con l'utilizzo della normale scansione.

L'indice sparso ha quindi una meno occupazione di spazio → posso tenere più parte dell'indice in RAM, con l'indice denso posso sapere se esiste un record con una certa chiave guardando solo l'indice (se cercassi l'indice 75 già dall'indice notiamo che il record non esiste).

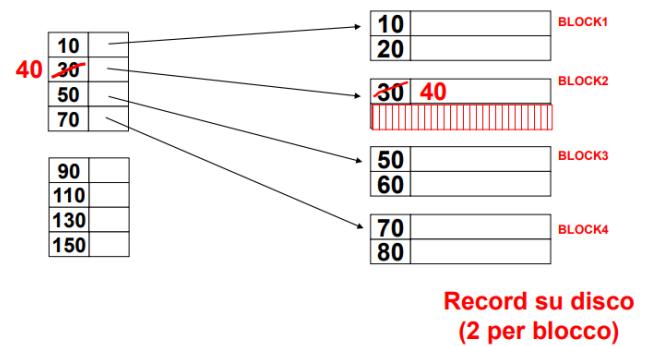
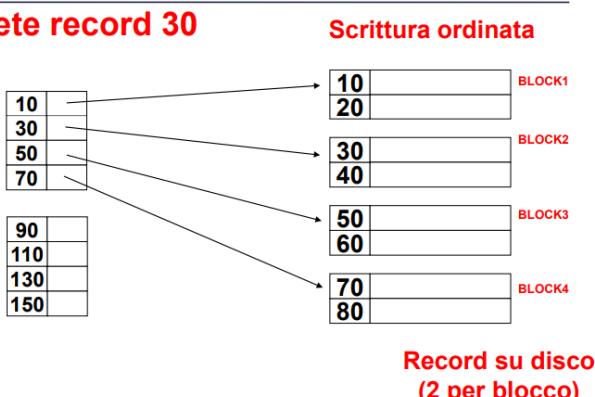
## Cancellazione da indice sparso

– delete record 40



## Cancellazione da indice sparso

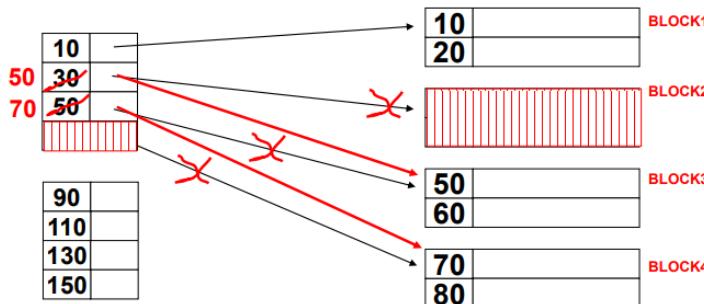
– delete record 30



Nel primo esempio vogliamo cancellare semplicemente il record 40 da un indice sparso, l'operazione risulta molto facile, nel secondo invece vogliamo cancellare il record 30, anche in questo caso l'operazione è facile però bisogna anche sostituire al 30 il record 40 e modificare il valore dell'indice,

Invece per eliminare sia i record 30 e 40, si rimuove l'intero blocco e si compatta l'indice; quindi, si rimuove semplicemente l'indice 30 e si portano in alto gli altri indici, per farlo bisogna anche cambiare i valori dei puntatori.

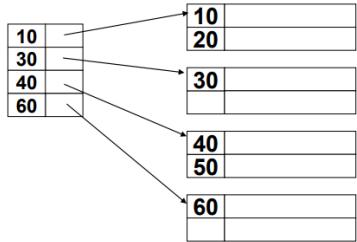
## – delete records 30 & 40



## Scrittura ordinata

Record su disco  
(2 per blocco)

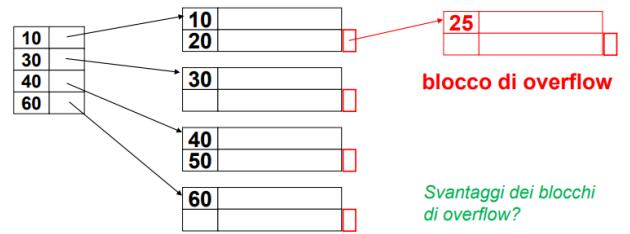
## – insert record 25



Scrittura ordinata

Cosa devo fare?

Cosa succede se non  
c'è spazio dopo il 30?



Scrittura ordinata

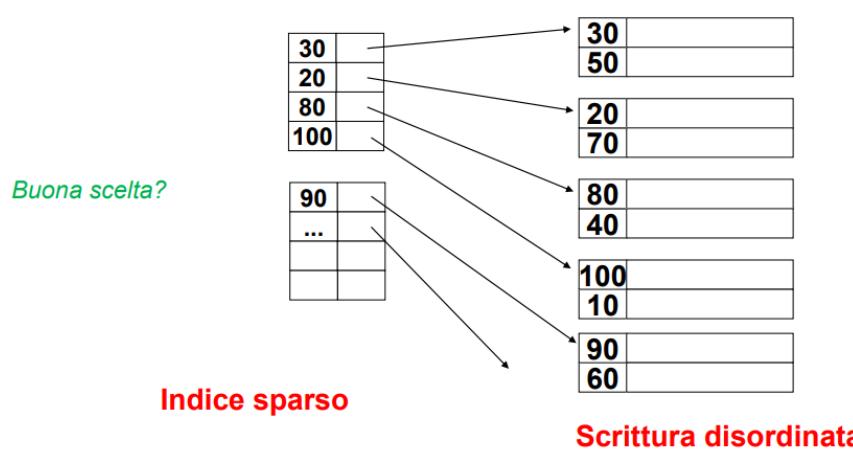
blocco di overflow  
Svantaggi dei blocchi  
di overflow?

In questo esempio vogliamo inserire il record 25, per farlo si utilizza il **blocco di overflow**, quindi si mantiene inalterato l'indice, e si crea un altro blocco con un puntatore dal primo verso il secondo. Quindi in questo caso si va a cercare il record tra gli indici 20 e 30, si accede al primo e si trova il record 20 che ha un puntatore ad un altro blocco in cui si trova il record 25.

Riprendendo il discorso sugli **indici primari**, quest'ultimi hanno diversi vantaggi e svantaggi:

- Vantaggi:
  - Semplici da implementare;
  - Efficienti per scansioni (= letture di sequenze contigue di record).
- Svantaggi:
  - Modifiche costose;
  - Difficile mantenere l'ordine.

Come si può indicizzare una **struttura disordinata**? Come questa:



Buona scelta?

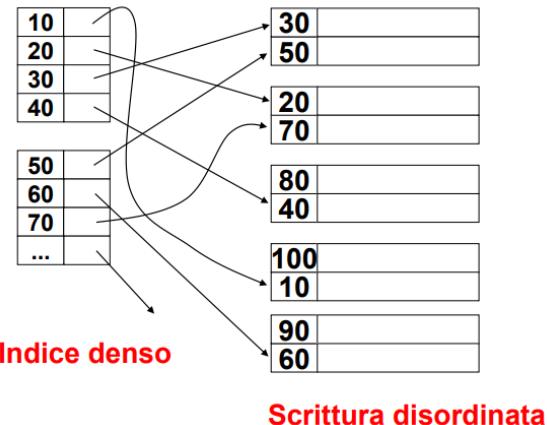
Indice sparso

Scrittura disordinata

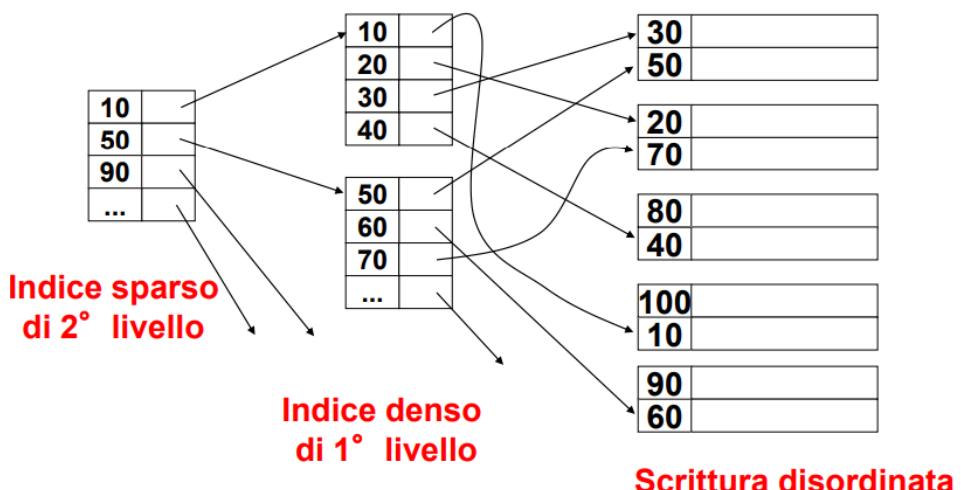
Si potrebbe pensare di utilizzare un **indice sparso**, ma non avendo un ordine dei dati sarebbe troppo confusionario e richiederebbe troppi accessi.

Ad esempio, se cercassimo il record 40, intuitivamente lo andremo a cercare dopo il 30, ma invece in quel blocco si trova il record 50. Quindi bisognerebbe verificare ogni record fino a che non si trova quello desiderato.

L'unica alternativa veramente applicabile è quella di utilizzare [l'indice denso](#), che, come detto prima, ha un'entrata per ogni singolo record:



Per ottimizzare quest'indice possiamo aggiungere dei [livelli](#), in cui nel 1º troviamo un indice denso, e nel 2º invece troviamo un indice sparso in modo tale da “mettere assieme” in un unico record un blocco.



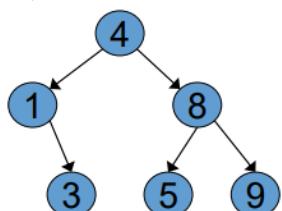
La caratteristica principale degli indici è l'accesso diretto sulla chiave in maniera efficiente, sia puntuale sia per intervalli.

Per quanto riguarda le [ricerche](#) su altri campi (non chiave), l'utilizzo del [indice](#) risulta inutile, a meno che non si crei un altro indice su quel particolare campo.

Le [modifiche](#) della chiave, gli [inserimenti](#), e le [eliminazioni](#) risultano [inefficienti](#) (come in tutte le strutture ordinate), esistono diverse tecniche per alleviare i problemi:

- blocchi di overflow;
- marcatura per le eliminazioni;
- riempimento parziale;
- riorganizzazioni periodiche.

Tutte le strutture di indice viste finora sono basate su [strutture ordinate](#) e quindi sono poco flessibili in presenza di elevata dinamicità. Per questo motivo nei DBMS si usano indici ad-hoc, più sofisticati, in particolare degli [indici dinamici multilivello](#): [B-tree](#), che sono [degli alberi di ricerca bilanciati](#).



In generale un [albero binario di ricerca](#) è un albero in cui ogni nodo è etichettato, e per ogni nodo il sottoalbero sinistro contiene solo etichette minori di quella del nodo e il sottoalbero destro etichette maggiori.

Il tempo di ricerca e l'inserimento hanno complessità pari alla [profondità](#), ed è logaritmico nel caso medio (assumendo un ordine di inserimento casuale).

In un [albero di ricerca di ordine P](#) (quindi non binario) ogni nodo ha (fino a) P figli e (fino a) P-1 etichette, ordinate.

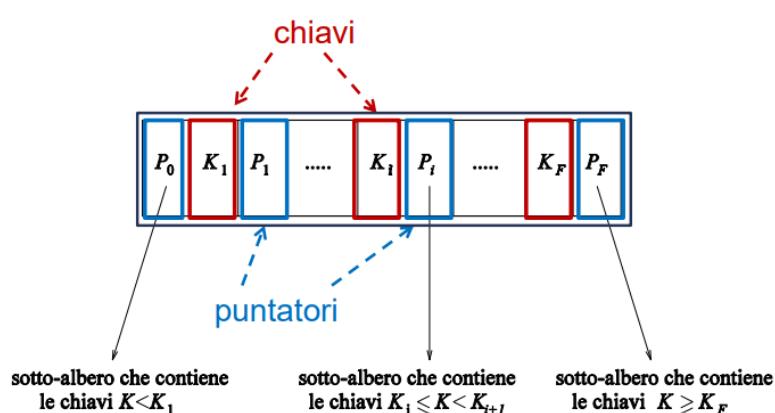
Nell'i-esimo sottoalbero abbiamo tutte etichette maggiori della (i-1)- esima etichetta e minori della i-esima. Ogni ricerca o modifica comporta la visita di un cammino radice → foglia.

In generale, nelle strutture fisiche, un **nodo** può corrispondere ad un **blocco di indici**.

Purtroppo, questo tipo di struttura risulta ancora troppo rigida, per questo motivo si utilizzano i [B-tree](#), che sono degli alberi di ricerca che vengono mantenuti bilanciati grazie a:

- [Riempimento parziale](#) (mediamente 70%): si cerca sempre di non riempire un nodo con troppe chiavi;
- [Riorganizzazioni](#) (locali) [in caso di sbilanciamento](#): in caso di albero sbilanciato non vi è la necessità di modificare l'intero albero, ma solo la sezione che lo sbilancia.

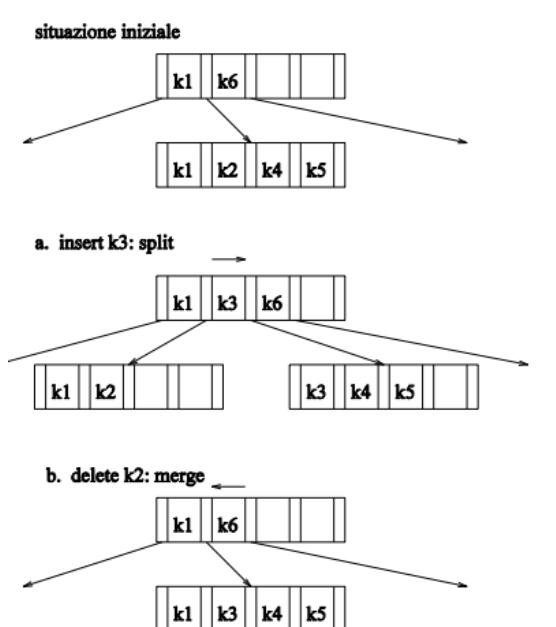
Un nodo del B-tree è così composto:



Ha una [serie di chiavi](#) che sono ordinate, e tra ogni coppia si trova un [puntatore](#). Il primo puntatore rimanda al sottoalbero che contiene le chiavi minori di K<sub>1</sub>, il secondo puntatore rimanda invece al nodo che contiene le chiavi fra K<sub>1</sub> e K<sub>2</sub>. L'ultimo puntatore rimanda invece al sottoalbero che contiene le chiavi [maggiori](#) dell'ultima, in questo caso K<sub>F</sub>.

Per quanto riguarda gli [inserimenti e le eliminazioni](#), esse sono precedute da [una ricerca fino ad una foglia](#) (nodo senza figli), se c'è posto nella foglia (quindi non si è ancora raggiunto il numero massimo di chiavi), si inserisce lì, altrimenti il **nodo** va suddiviso in due parti, con un puntatore in più per il genitore, se non c'è posto, si sale ancora.

Questo algoritmo garantisce che il **riempimento** rimanga sempre superiore al 50%.



In questo esempio abbiamo una situazione iniziale in cui il nostro nodo del B-tree ha solo due chiavi K<sub>1</sub> e K<sub>6</sub>, e tre puntatori, se noi volessimo inserire la chiave K<sub>3</sub>, come visto prima, si parte dalla radice e si va a cercare il nodo foglia in cui inserirlo.

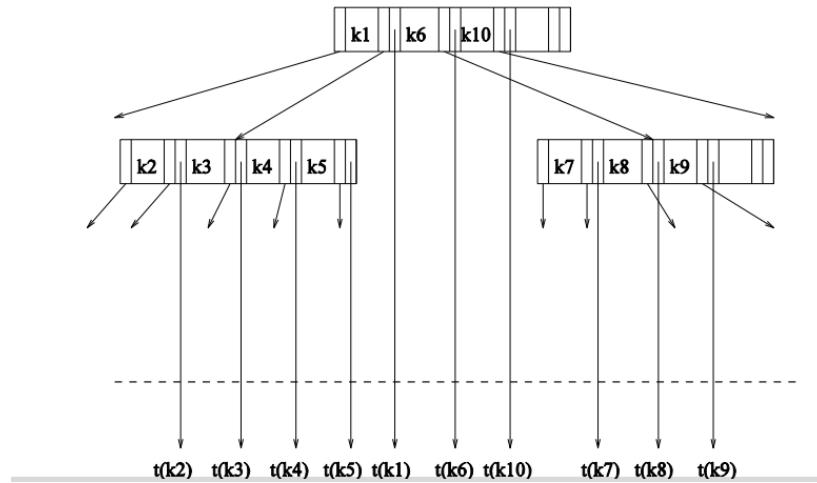
In questo caso lo dovremo inserire tra le chiavi K<sub>2</sub> e K<sub>4</sub> ma quel nodo è già completo, quindi bisogna [dividerlo](#) (split). Per farlo si porta in alto il valore centrale che è proprio K<sub>3</sub> (nella radice), e si creano quindi due nodi, il primo con K<sub>1</sub> e K<sub>2</sub>, e un secondo con i nodi K<sub>3</sub>, K<sub>4</sub>, K<sub>5</sub>.

Invece se volessimo [eliminare](#) la chiave k<sub>2</sub> succederebbe questo:

Considerando che il nodo in cui si trova K<sub>2</sub> è quasi vuoto, e rimuovendo la chiave rimarrebbe solo K<sub>1</sub>, si utilizza il [merge](#), con cui si rimuove K<sub>2</sub>, e si riuniscono i due nodi, ottenendo un unico nodo contenente K<sub>1</sub>, K<sub>3</sub>, K<sub>4</sub> e K<sub>5</sub>.

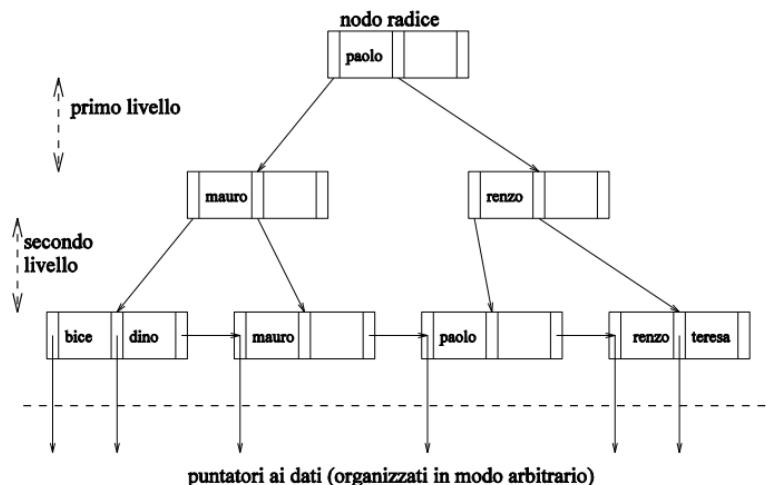
Nei B-tree i nodi intermedi possono avere dei puntatori direttamente ai dati, nell'esempio sotto possiamo notare che dopo la chiave K1 vi è un primo puntatore, ma anche un secondo, che riporta proprio alla tupla in cui è presente quella chiave.

## Un B-tree



Ma esistono anche i [B+ tree](#), in cui le foglie sono collegate in una lista, e questo li rende ottimi per le ricerche sugli intervalli e vengono molto utilizzati nei DBMS. Con questo albero si ha che per ogni foglia si ha un puntatore che porta alla foglia successiva.

## Un B+ tree



Se volessimo ricercare i nomi che iniziano da M e R, si parte dalla radice e si vede Paola, per cui la P è > di M (perché la P viene dopo la M), quindi si va a ricercare a sinistra, e si trova Mauro, e per trovare un nome che inizia per R si va a cercare a destra perché M è < di R. Questo tipo di albero ci permette di non dover risalire l'albero ogni volta, grazie al fatto che si trovano i puntatori alla foglia successiva in ogni foglia.

L'efficienza dei B-tree è evidente quando il numero delle chiavi per blocco è grande, in quanto lo splitting e il merging sono poco frequenti, e in genere limitati a piccole porzioni dell'albero, e quando le chiavi in sé sono piccole e compatte.

La [profondità](#) tipica è di 3-4 livelli, eccetto per i DB molto grandi. Il [numero di letture](#) è uguale alla profondità dell'albero + 1 (quest'uno serve per poter leggere il record su disco).

# HASHING, PROCESSING, E OTTIMIZZAZIONE DELLE QUERY

L'obiettivo dell'**hashing** è l'accesso diretto ad un insieme di record sulla base del valore di un campo che viene detto impropriamente "chiave", anche se non è detto sia identificante.

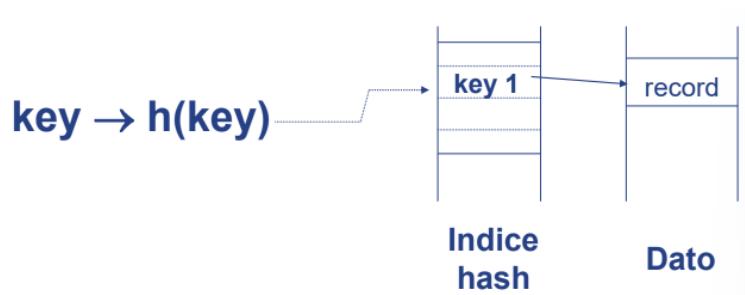
Una **funzione hash** è una funzione che associa ad ogni valore della chiave un "indirizzo", in uno spazio di dimensione leggermente superiore al necessario.

La funzione hash è costituita in questo modo:

$$h : C \rightarrow I$$

Dove **C** è l'insieme delle possibili chiavi (**dominio**) e **I** è l'insieme degli indirizzi (**codominio**). Un aspetto importante è che la cardinalità del dominio è molto maggiore rispetto alla cardinalità del codominio; quindi, questo significa che nel caso delle matricole non avremo un indirizzo per ogni matricola ma potrebbe capitare che a diverse matricole corrisponda lo stesso indirizzo:

**Cardinalità( I ) << Cardinalità( C )**



Il **funzionamento** è questo: partiamo dalla nostra chiave, applichiamo la funzione hash su quest'ultima e otteniamo un indice hash che è un puntatore al record (che si trova in memoria) che stavamo cercando.

Una **buona funzione hash** deve distribuire i valori delle chiavi negli indirizzi in modo **uniforme**, rispettando questa condizione la possiamo utilizzare per trovare subito la posizione di un record con una determinata chiave.

Il problema principale di questo metodo sono le **collisioni**: dato che il numero di possibili chiavi è molto maggiore del numero di possibili indirizzi, la funzione hash **non può essere iniettiva**; quindi, a più chiavi potrebbe capitare che corrisponda lo stesso indirizzo, questo per l'appunto causa una collisione.

- **Funzione hash h:**

$$h(k) = k \bmod 50$$

- 40 record

- tavola hash con 50 posizioni:

- 1 collisione a 4
- 2 collisioni a 3
- 5 collisioni a 2

M	M mod 50
60600	0
66301	1
205751	1
205802	2
200902	2
116202	2
200604	4
66005	5
116455	5
200205	5
201159	9
205610	10
201260	10
102360	10
205460	10
205912	12
205762	12
200464	14
205617	17
205667	17

M	M mod 50
200268	18
205619	19
210522	22
205724	24
205977	27
205478	28
200430	30
210533	33
205887	37
200138	38
102338	38
102690	40
115541	41
206092	42
205693	43
205845	45
200296	46
205796	46
200498	48
206049	49

## ESEMPIO

In questo esempio vogliamo indicizzare le matricola, la prima cosa da fare è trovare la funzione hash, che in questo caso è  $k$  (matricola) mod 50.

La cardinalità della funzione hash è 50, quindi ci può restituire al massimo 50 indirizzi diversi. Possiamo notare che ci sono diverse collisioni.

Per ridurre le probabilità che si verifichino le collisioni possiamo:

- Usare una “buona” funzione hash (che distribuisca in modo causale e uniforme);
- Aumentare lo spazio ridondante (spazi vuoti).

Per affrontarle realmente, quindi fare in modo che proprio non si verifichino ci sono 2 soluzioni:

- Posizioni successive disponibili;
- Tabella di overflow.

## ESEMPI

- 40 record
- tavola hash con 50 posizioni:
  - 1 collisione a 4
  - 2 collisioni a 3
  - 5 collisioni a 2
  - Se 1 blocco = 1 record:  
numero medio di accessi: 1,425

M	M mod 50	M	M mod 50
60600	0	200268	18
66301	1	205619	19
205751	1	210522	22
205802	2	205724	24
200902	2	205977	27
116202	2	205478	28
200604	4	200430	30
66005	5	210533	33
116455	5	205887	37
200205	5	200138	38
201159	9	102338	38
205610	10	102690	40
201260	10	115541	41
102360	10	206092	42
205460	10	205693	43
205912	12	205845	45
205762	12	200296	46
200464	14	205796	46
205617	17	200498	48
205667	17	206049	49

In questo esempio come prima abbiamo 40 record e una tavola hash con 50 posizioni, e abbiamo che in un blocco ci sta solamente un record.

Dato che in un blocco ci sta un solo record, per risolvere questo problema, possiamo utilizzare dei puntatori all'interno dei record che mandano al secondo record che era stato indicizzato nella stessa posizione. Purtroppo, questa soluzione non è ottimale perché il numero medio di accessi è molto maggiore, circa 1,425.

Anche in questo esempio abbiamo sempre 40 record con una tavola hash con 50 posizioni, la differenza con il precedente è che in ogni blocco ci stanno 10 record.

In questo caso la situazione migliora di molto, anche se si verificano ancora delle collisioni ma il numero medio di accessi è di circa 1,05

## File hash con fattore di blocco 10; 5 blocchi con 10 posizioni ciascuno

| M mod 5 |
|---------|---------|---------|---------|---------|
| 60600   | 66301   | 205802  | 200268  | 200604  |
| 66005   | 205751  | 200902  | 205478  | 201159  |
| 116455  | 115541  | 116202  | 210533  | 200464  |
| 200205  | 200296  | 205912  | 200138  | 205619  |
| 205610  | 205796  | 205762  | 102338  | 205724  |
| 201260  |         | 205617  | 205693  | 206049  |
| 102360  |         | 205667  | 200498  |         |
| 205460  |         | 210522  |         |         |
| 200430  |         | 205977  |         |         |
| 102690  |         | 205887  |         |         |
| 205845  |         |         |         |         |
|         |         | 206092  |         |         |

40 record - tavola hash con 50 posizioni:  
file hash con fattore di blocco 10  
5 blocchi con 10 posizioni ciascuno:  
Due soli overflow - numero medio di accessi: 1,05

Per gestire la crescita di dimensione dei dati si utilizza l'**hashing dinamico**, che consiste nell'utilizzare solo i primi *i* bit dell'output della funzione hash, al posto di utilizzare tutti i *b* bit:

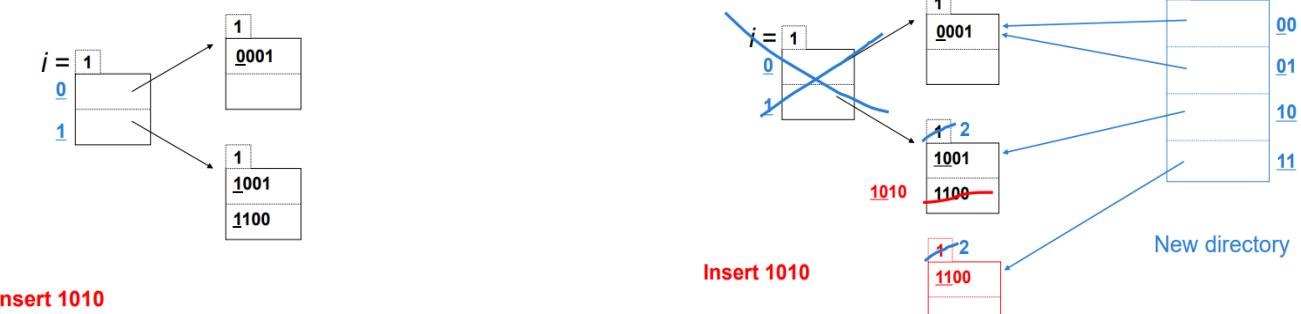


la dimensione di *i* cresce col tempo...

Quindi pian piano che la tabella cresce e ci si accorge che si stanno per verificare delle collisioni, si inizia ad aumentare i bit di *i*. Quest'algoritmo funziona anche all'inverso, cioè che se la tabella dovesse diminuire di dimensione si diminuirebbero di conseguenza i bit di *i* considerati.

## ESEMPIO 1

Esempio:  $h(k)$  restituisce 4 bit; 2 chiavi per blocco

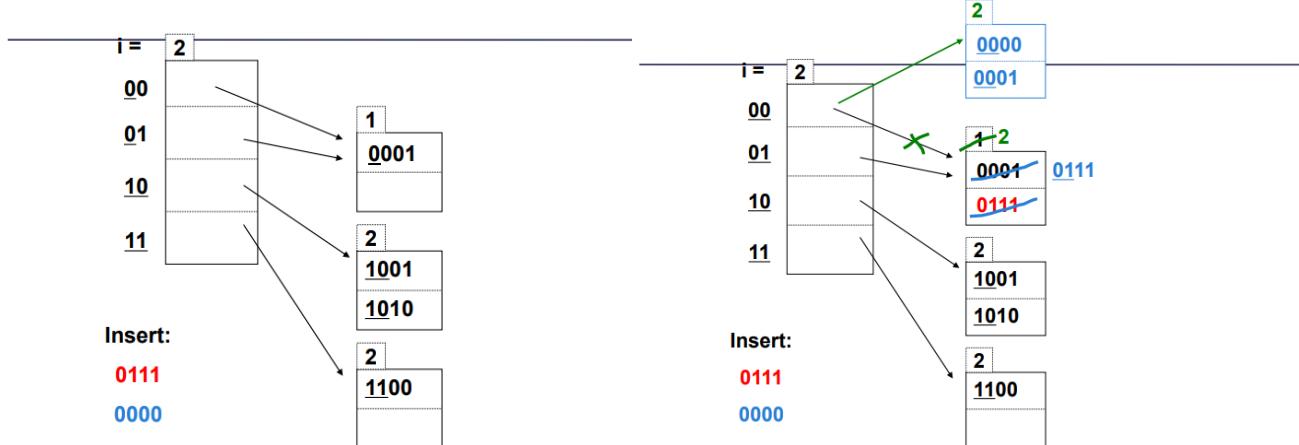


In questo esempio vogliamo inserire il valore **1010**, che ci è stato restituito dalla funzione hash, e possiamo memorizzare solo 2 chiavi per blocco. Per poterlo fare consideriamo il primo bit che è 1, i due blocchi hanno rispettivamente come primo bit 0 e come primo bit 1, quindi il nostro numero dovrà essere inserito nel secondo blocco.

Questo blocco però è già pieno, per poterlo inserire consideriamo i primi 2 bit di quei 3 numeri, quei numeri che iniziano con 10 verranno inseriti nel blocco originario; invece, il numero che inizia con 11 verrà inserito in un nuovo blocco.

Infine, dobbiamo aggiornare la tabella con gli indici della funzione hash, dove avremo che i valori che iniziano con 00 e 01 verranno mappati nel primo blocco, i valori che iniziano con 10 nel blocco 2 e così via.

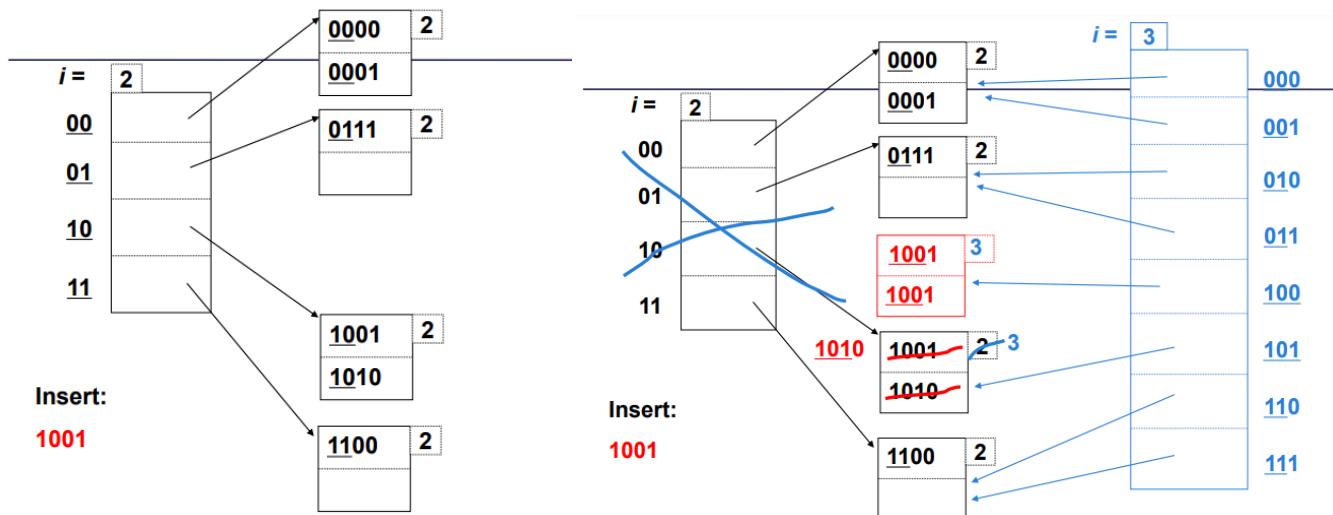
## ESEMPIO 2



In questo esempio vogliamo inserire i valori **0111** e **0000**, il primo viene facilmente inserito nel primo blocco, ma anche il secondo dovrebbe essere inserito in quel blocco, ma ora risulta pieno e quindi si verificherebbe una collisione.

Per risolvere consideriamo i primi 2 bit e creiamo un nuovo blocco, in quest'ultimo inseriremo 0111, nel altro blocco inseriremo 0000.

### ESEMPIO 3



In quest'ultimo esempio dobbiamo inserire il valore 1001, che dovrebbe essere inserito nel blocco contenente 1001 e 1010, ma notiamo che 1001 è già presente. In questa situazione si ha una collisione perché due chiavi vengono mappate con lo stesso valore, per risolverlo si utilizzano i puntatori all'interno dei record.

Per poter capire dove inserirlo non basta considerare i primi 2 bit perché avremo l'uguaglianza fra 1001, 1001 e 1010, quindi controlliamo i primi 3 bit e inseriamo nel primo blocco i valori 1001 e nel secondo blocco il valore 1010.

L'**hashing dinamico** o **file hash** è l'organizzazione più efficiente per l'accesso diretto basato su valori della chiave con condizioni di uguaglianza, ma non è efficiente per ricerche basate su intervalli; quindi, posso effettuare delle ricerche solo per un dato esatto. Il **costo medio** è di poco superiore all'unità, il caso peggiore è molto costoso ma improbabile.

L'**indice bitmap** è più adatto per colonne con un basso numero di possibili valori diversi, e funziona in questo modo:

Tabella		pointers	Indice Bitmap(Stato_Civile)	
Person_ID	Marital_Status		Single	Married
1001	Single	←	1	0
1002	Single	←	1	0
1003	Married	←	0	1
1004	Single	←	1	0
1005	Married	←	0	1
1006	Single	←	1	0
1007	Married	←	0	1
1008	Married	←	0	1
1009	Single	←	1	0

In questo esempio l'indice bitmap avrà una colonna per ogni valore del Marital\_Status, e in ogni colonna avrà un puntatore al record corrispondente.

La caratteristica principale dell'indice bitmap è che restituisce una **tabella molto compatta**.

Numero di colonne = numero di valori differenti

**Tabella**

Person_ID	Age_Range
1001	18-34
1002	65+
1003	65+
1004	50-65
1005	35-49
1006	35-49
1007	65+
1008	35-49
1009	50-65

**Indice Bitmap (Age\_Range)**

18-34	35-49	50-65	65+
1	0	0	0
0	0	0	1
0	0	0	1
0	0	1	0
0	1	0	0
0	1	0	0
0	0	0	1
0	1	0	0
0	0	1	0

Un suo utilizzo molto efficace è quello di effettuare query su molti campi, ognuno con pochi valori possibili, non è però adatto per dati molto dinamici, cioè quando si fanno delle insert, update e delete molto frequenti perché un'operazione su un dato può "bloccare" migliaia di altre operazioni su altri dati della stessa tabella (scritture concorrenti sull'indice bitmap).

*"Trova tutti gli uomini anziani non sposati"*

**Tabella**

Person_ID	Age_Range
1001	18-34
1002	65+
1003	65+
1004	50-65
1005	35-49
1006	35-49
1007	65+
1008	35-49
1009	50-65

**Indice Bitmap (varie colonne)**

18-34	65+	Married	Male
1	0	0	1
0	1	0	1
0	1	1	0
0	0	0	1
0	0	1	0
0	0	0	1
0	1	1	0
0	0	1	0
0	0	0	1

Per definire gli indici con i comandi SQL si utilizza questa sintassi:

- **create [unique] index IndexName  
on TableName(AttributeList)**

Inserendo **unique** stiamo specificando che AttributeList è una superchiave. Per eliminare un indice questa è la sintassi:

**drop index IndexName**

Un esempio è questo, dove stiamo creando un indice che "indicizza" in base al cognome e alla città:

- **create index NomeIndice  
on Impiegato(Cognome, Citta)**

Creare un indice ha senso quando si ha la necessità di utilizzare una query molto spesso e se si vuole frequentemente recuperare meno del 15% delle righe di una grande tabella. Solitamente è anche molto utile indicizzare le colonne usate per i join in quanto le chiavi primarie e i campi "unique" hanno automaticamente indici, ma può essere utile creare indici per le chiavi esterne.

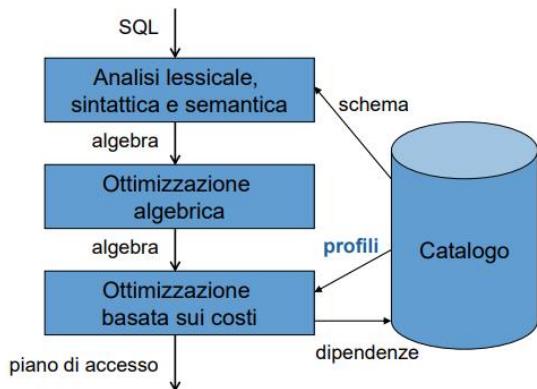
Infine, è utile per indicizzare le colonne i cui valori sono relativamente univoci (eccetto che per gli indici bitmap).

## ESECUZIONE E OTTIMIZZAZIONE DELLE QUERY

Il **query processor** o **ottimizzatore** è un modulo del DBMS che si occupa di decidere in quale maniera eseguire la query nel modo più efficiente possibile. Le query SQL sono espresse ad alto livello, il linguaggio SQL è (di base) un linguaggio dichiarativo.

L'algebra relazionale offre un approccio procedurale, l'ottimizzatore sceglie la strategia realizzativa a partire dall'istruzione SQL.

### Esecuzione delle query



In input si ha una stringa SQL, di cui viene fatta un'**analisi lessicale, sintattica e semantica** da un determinato software. Se la query è corretta viene trasformata nell'espressione algebrica che poi verrà ottimizzata in base ai costi delle diverse operazioni.

Il termine **ottimizzazione** è improprio perché il processo utilizza euristiche (insieme di strategie). Si basa sulla nozione di **equivalenza**: due espressioni sono equivalenti se producono lo stesso risultato qualunque sia l'istanza attuale della base di dati.

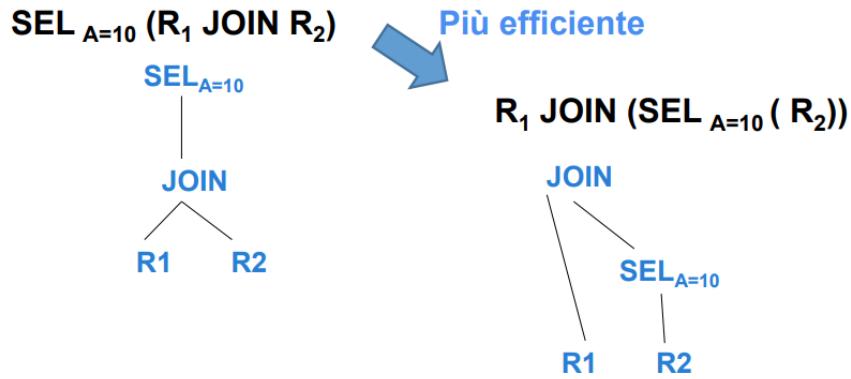
I DBMS cercano di eseguire espressioni equivalenti a quelle date, ma meno "costose".

L'**euristica fondamentale** afferma che "le selezioni e proiezioni il più presto possibile", questo per ridurre le dimensioni dei risultati intermedi. Bisogna quindi seguire le regole "**push selections down**" e "**push projections down**".

Un esempio di **push selections down** è questo: assumiamo che A sia un attributo di R2 →

$$\begin{array}{c} \text{SEL A=10 (R1 JOIN R2)} \\ \leftrightarrow \\ \text{R1 JOIN (SEL A=10 (R2))} \end{array}$$

La seconda forma riduce in modo significativo la dimensione del risultato intermedio (e quindi il costo dell'operazione), questo perché applicare la select solo su R2 è molto meno "costoso" rispetto a farlo sulla tabella risultante del JOIN che è molto più grande.



Le query vengono rappresentate internamente tramite **alberi** dove le foglie sono i dati e i nodi intermedi sono gli operatori.

I DBMS implementano gli operatori dell'algebra per mezzo di **operazioni di basso livello** che possono implementare vari operatori "in un colpo solo". Gli operatori fondamentali sono quelli di **scansione** e di **accesso diretto**, a livello più alto si ha l'**ordinamento** e il **join**.

Un **accesso diretto** è possibile se si utilizzano degli **indici** o delle **strutture hash**; invece, la **scansione** è molto efficiente se si usa un **B+ tree**.

Il **JOIN** è l'operazione più costosa e più frequente, ci sono diversi metodi per effettuarlo:

1. Nested-loop (chiamato anche "iteration join");
2. Merge-join (chiamato anche "merge scan");
3. Hash-join.

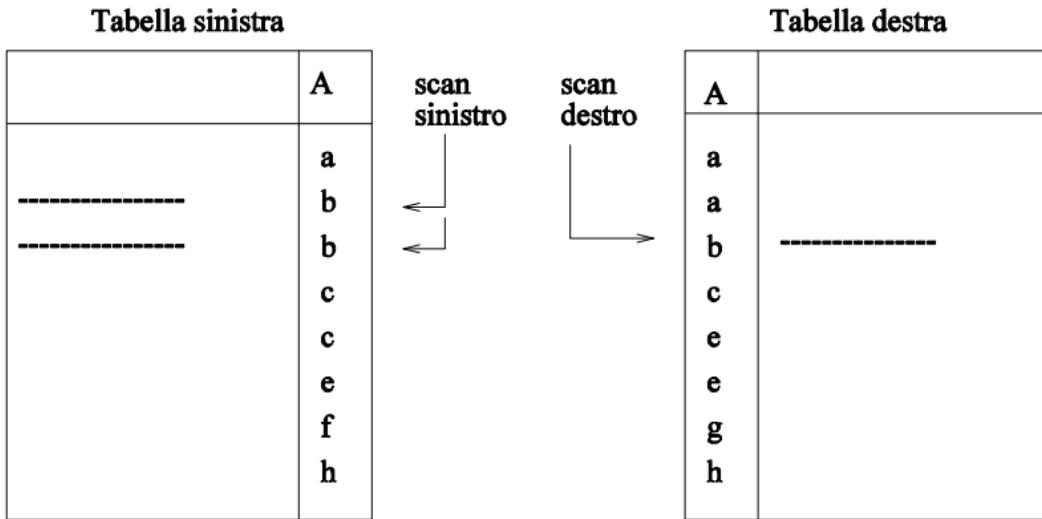
Il **Nested-loop** è così realizzato:

```
(concettualmente) R1 JOIN R2 ON R1.C = R2.C:
for each r ∈ R1 do
    for each s ∈ R2 do
        if r.C = s.C then output <r,s> pair
```

Abbiamo quindi due **for innestati** con i quali si scorrono tutti gli elementi di R1 e di R2, finché non si trova la corrispondenza fra le chiavi. La **cardinalità** qui è il **prodotto cartesiano fra le due tabelle**.

Il **Merge-join** si basa sul fatto di avere una struttura ordinata su una chiave, o che sia indicizzata con un **B+ tree**, ed è realizzato così:

```
(concettual.) R1 JOIN R2 ON R1.C = R2.C:
(1) Ordinare R1 e R2 su C, o indicizzarli con B+ tree
(2) i ← 1; j ← 1;
    while (i ≤ #T(R1)) ∧ (j ≤ #T(R2)) do
        if R1{ i }.C = R2{ j }.C then {
            outputTuples(R1{ i }.C); i ← i+1; j ← j+1 }
        else if R1{ i }.C > R2{ j }.C then { j ← j+1 }
        else if R1{ i }.C < R2{ j }.C then { i ← i+1 }
```



Si fa uno **scan sinistro** che parte dalla prima riga e vede il valore a, passa alla seconda tabella e incontra il valore a e restituisce la coppia, scende e trova un'altra a e restituisce un'altra coppia, scende ancora e trova b; quindi, capisce che le a sono finite; quindi, torna nella tabella sinistra e riparte da b e così via. La **cardinalità** qui è della **somma fra le cardinalità delle due tabelle**, il caso estremo è quando si ha sempre lo stesso valore nella tabella di sinistra e di destra.

L'**Hash join** è un algoritmo in cui indicizzo i campi tramite l'utilizzo dell'hash, si raggruppano i valori del campo del join in due buckets:

Hash function h, range 0 → k.

- Buckets per R1: G0, G1, ... Gk;
- Buckets per R2: H0, H1, ... Hk.

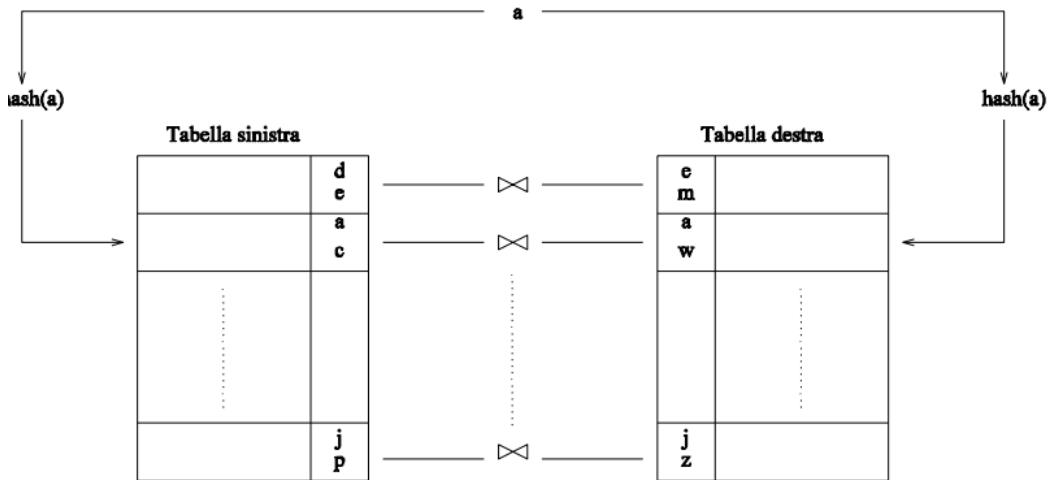
Algoritmo per R1 JOIN R2 ON R1.C = R2.C:

1. Fare Hash delle tuple di R1 in G buckets (in base al valore di C);
2. Fare Hash delle tuple di R2 in H buckets (in base al valore di C);
3. For i = 0 to k do: confrontare le tuple nei bucket Gi e Hi

#### Simple example: hash even/odd

R1	R2	Buckets			
2	5	Pari:	2 4 8	4 12 8 14	
4	4		R1	R2	
3	12	Dispari:	3 5 9	5 3 13 11	
5	3				
8	13				
9	8				
	11				
	14				

In questo caso stiamo suddividendo i valori di R1 e R2 in due Buckets che li divide in **numeri pari e dispari**, questi buckets ovviamente possono essere sfruttati e utilizzati per il join, perché sarà più facile ricercare i valori.



Graficamente possiamo vedere che abbiamo diversi bucket, ad esempio d e si trovano in uno stesso bucket perché hanno lo stesso valore di hash, l'Hash join funziona come se si stessero effettuando degli Nested-loop fra le coppie dei bucket per cui valori restituiti dalla funzione hash sono gli stessi.

Il [processo di ottimizzazione](#) è suddiviso in diversi step:

- Si costruisce un albero di decisione con le varie alternative ("piani di esecuzione");
- Si valuta il costo di ciascun piano;
- Si sceglie il piano di costo minore;

Il cosiddetto "ottimizzatore" trova di solito una "buona" soluzione, non necessariamente l'ottimo; l'ottimizzatore non dev'essere troppo complesso o si perde il vantaggio dell'ottimizzazione!