

BASI-DI-DATI-Lezioni_book

Lezioni

-  [1-Introduzione](#)
 -  [Algebra Relazionale](#)
 -  [DBMS](#)
 -  [DDL](#)
 -  [Normalizzazione](#)
 -  [SQL](#)
-

1-Introduzione

1-Introduzione

Obiettivi dell'IA (I):

- Modellare "fedelmente" l'essere umano:
 1. Agire umanamente: *Test di Turing*
 2. Pensare umanamente: modelli cognitivi per descrivere il funzionamento della mente umana
- Raggiungere risultati ottimali
 1. Pensare razionalmente: studio di facoltà mentali tramite modelli computazionali, e.g. la *tradizione logicista* (per lo studio delle leggi/processi del pensiero che si ritiene «guidino» la mente)
 2. Agire razionalmente: **Agenti razionali**, automazione del comportamento intelligente

True sempre vero; False sempre falso

$P \wedge Q$, vero se P e Q sono entrambi veri

$P \vee Q$, vero se P oppure Q , o entrambi, sono veri

$\neg P$ vero se P è falso

$P \Rightarrow Q$, vero se P è falso oppure Q è vero

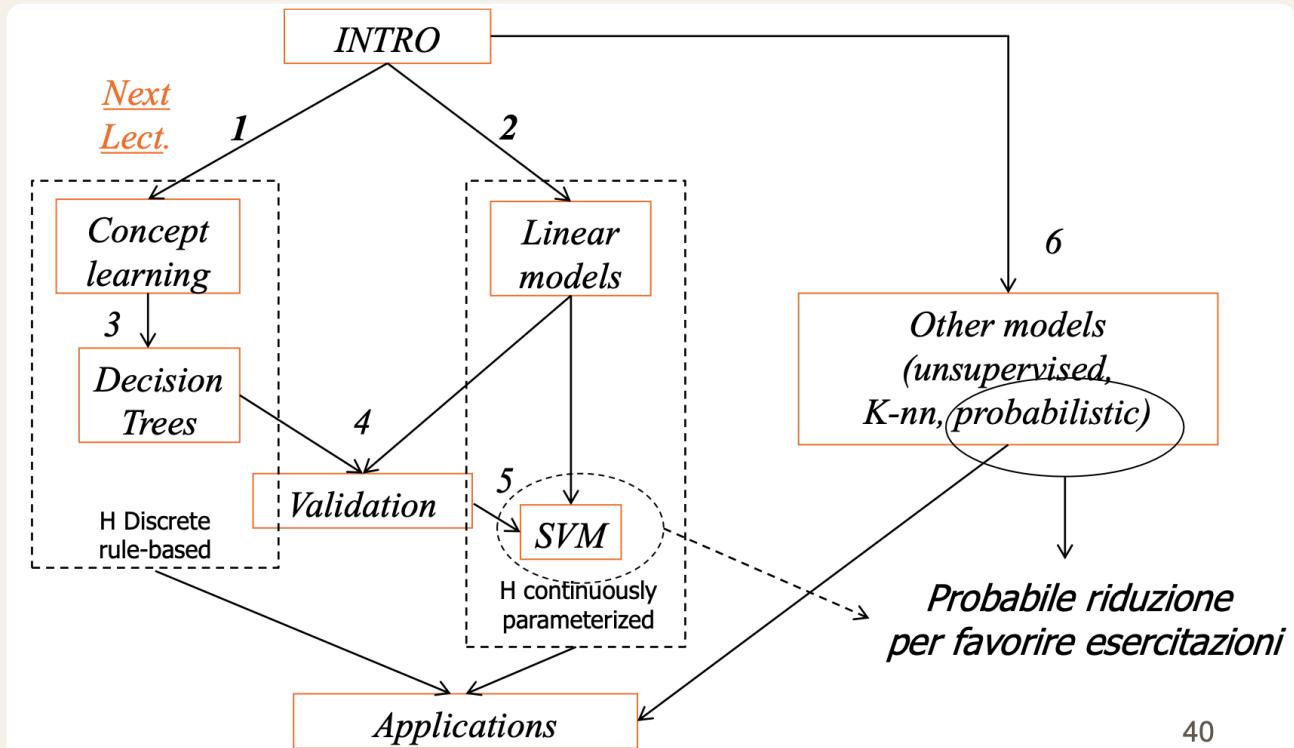
$P \Leftrightarrow Q$, vero se entrambi veri o entrambi falsi

$$\forall x \neg P(x) \equiv \neg \exists x P(x) \quad | \quad \neg P \wedge \neg Q \equiv \neg(P \vee Q)$$

$$\neg \forall x P(x) \equiv \exists x \neg P(x) \quad | \quad \neg(P \wedge Q) \equiv \neg P \vee \neg Q$$

$$\forall x P(x) \equiv \neg \exists x \neg P(x) \quad | \quad P \wedge Q \equiv \neg(\neg P \vee \neg Q)$$

$$\neg \forall x \neg P(x) \equiv \exists x P(x) \quad | \quad P \vee Q \equiv \neg(\neg P \wedge \neg Q)$$



minimi funzione errore \Rightarrow gradiente = 0

$$w_I = \frac{\sum x_p y_p - \frac{1}{l} \sum x_p \sum y_p}{\sum x_p^2 - \frac{1}{l} (\sum x_p)^2} = \frac{\text{Cov}[x, y]}{\text{Var}[x]}, \quad w_0 = \bar{y} - w_I \bar{x}$$

$p: 1 \rightarrow l$

$$\frac{1}{l} \sum_{p=1}^l y_p \quad \frac{1}{l} \sum_{p=1}^l x_p$$

Where l is num. of examples

Hence we omit p for x

Basic rules:

$$\begin{aligned} \frac{\partial}{\partial w} k &= 0, \frac{\partial}{\partial w} w = 1, \frac{\partial}{\partial w} w^2 = 2w \\ \frac{\partial(f(w))^2}{\partial w} &= 2f(w) \frac{\partial(f(w))}{\partial w} \end{aligned}$$

Der. sum = sum of der.

$$\frac{\partial E(\mathbf{w})}{\partial w_i} = \frac{\partial(y - h_{\mathbf{w}}(x))^2}{\partial w_i} =$$

$$= 2(y - h_{\mathbf{w}}(x)) \frac{\partial(y - h_{\mathbf{w}}(x))}{\partial w_i} = 2(y - h_{\mathbf{w}}(x)) \frac{\partial(y - (w_1 x + w_0))}{\partial w_i}$$

$$\frac{\partial E(\mathbf{w})}{\partial w_0} = -2(y - h_{\mathbf{w}}(x))$$

$$\frac{\partial E(\mathbf{w})}{\partial w_1} = -2(y - h_{\mathbf{w}}(x)) \cdot x$$

$$\frac{\delta(y - (w_1 x + w_0))}{\delta w_0} = 0 - (0 + 1) = -1$$

$$\frac{\delta(y - (w_1 x + w_0))}{\delta w_1} = 0 - (0 - x + 0) = -x$$

Algebra Relazionale

Algebra Relazionale

DDL: data definition language

- operazioni di creazione, cancellazione e modifica di tabelle
DML: data manipulation language
- Data Query Language - Query o interrogazione della BD
- Aggiornamento dei dati - Inserimento, cancellazione e modifica di dati

È possibile applicare unione, intersezione, differenza, ... , solo a tabelle definite sugli stessi attributi

Unione

Laureati

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

Matricole

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

Laureati \cup Matricole

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

Laureandi

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

Laureati

Codice	Nome	Età
1100	Viola	30
2000	Neri	22
2001	Moro	34

Laureandi \cup Laureati

??

Impossibile unire perché hanno attributi diversi, se rinominiamo {Id} e {Codice} in {Matricola} allora possiamo farlo.

$$\rho_{\text{Matricola} \leftarrow \text{Id}}(\text{Laureandi}) \rho_{\text{Matricola} \leftarrow \text{Codice}}(\text{Laureati})$$

$$T' = \rho_{A \leftarrow A'}(T)$$

Differenza

La differenza tra due relazioni R e S (R-S) produce una nuova relazione che contiene tutte le tuple che sono presenti in R ma non in S. Come per l'unione, le relazioni devono essere compatibili, ovvero definite sugli stessi attributi. Il risultato mantiene lo schema di R.

R		
A	B	C
a1	b1	c1
a1	b1	c2
a2	b1	c1
a3	b1	c1

S		
A	B	C
a1	b1	c1
a1	b2	c2

R - S =		
A	B	C
a1	b1	c2
a2	b1	c1
a3	b1	c1

Proiezione

$$\pi_{A1 \dots An}(R)$$

Restituisce una tabella contenente solo le n-uple distinte di R proiettate sugli attributi selezionati.

R		
A	B	C
a1	b1	c1
a1	b1	c2
a2	b1	c1
a3	b1	c1

$\pi_A(R) =$	
A	
a1	
a2	
a3	

Impiegati

Matricola	Cognome
7309	Neri
5998	Neri
9553	Rossi
5698	Rossi

$$\pi_{\text{Matricola}, \text{Cognome}}(\text{Impiegati})$$

Se {Matricola} è l'unico attributo che differisce tra due record mentre gli altri sono identici, una proiezione su {Cognome} come nell'esempio sopra restituirà solo le prime n-uple distinte trovate, in questo caso:

Cognome

Neri

Rossi

Se X è una superchiave di R , allora $\pi_x(R)$ contiene esattamente tante n-uple quante R

Se X non è superchiave, nella proiezione i valori ripetuti verranno eliminati e contati una sola volta

Selezione

<u>R</u>		
<u>A</u>	<u>B</u>	<u>C</u>
a1	b1	c1
a1	b1	c2
a2	b1	c1
a3	b1	c1

$$\sigma_{A='a_1'}(R) = \underline{\underline{\begin{array}{ccc} A & B & C \\ a1 & b1 & c1 \\ a1 & b1 & c2 \end{array}}}$$

La selezione è un'operazione che permette di estrarre le tuple di una relazione che soddisfano una determinata condizione. Il risultato mantiene lo schema della relazione di partenza, filtrando solo i record che rispettano i criteri specificati. La selezione viene indicata con il simbolo σ (sigma) seguito dalla condizione di selezione.

EX.

Quali sono gli impiegati che guadagnano più di 50?

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

$\sigma_{\text{Stipendio} > 50} (\text{Impiegati})$

Quali sono gli impiegati che guadagnano più di 50 e vivono a Milano?

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

Quali sono gli impiegati che hanno lo stesso cognome della filiale in cui lavorano?

$\sigma_{\text{Cognome} = \text{Filiale}} (\text{Impiegati})$

La condizione può diventare vera solo per valori non NULL

Per riferirsi ai valori NULL esistono forme apposite di condizioni:

- **IS NULL**
- **IS NOT NULL**

Bisogna ricordarsi di considerare i valori NULL all'interno delle nostre tabelle.

L'unione di tutti i casi possibili di un attributo, ma non prendendo in considerazione i casi in cui è NULL, potrebbe dare un risultato non previsto.

$$\sigma_{\text{Età} < 30} (\text{Persone}) \cup \sigma_{\text{Età} \geq 30} (\text{Persone}) \cup \sigma_{\text{Età IS NULL}} (\text{Persone}) = \text{Persone}$$

Proiezione e Selezione

$$\pi_{\text{Matricola}, \text{Cognome}} (\sigma_{\text{Provincia} = \text{Pisa}} (\text{Studenti}))$$

Si possono combinare per ottenere i risultati precisi che stiamo cercando. Qua troveremo Nome e Matricola degli studenti che vengono da Pisa.

Studenti			
Nome	Matricola	Provincia	Nascita
Isaia	71523	Pisa	1998
Rossi	67459	Lucca	1999
Bianchi	79856	Livorno	1998
Bonini	75649	Pisa	1999

Esami			
Materia	Candidato*	Data	Voto
BD	71523	12/01/2018	28
BD	67459	15/09/2019	30
IA	79856	25/10/2020	30
BD	75649	27/06/2018	25
IS	71523	10/10/2019	18

$\pi_{\text{Nome}, \text{Matricola}} (\sigma_{\text{Provincia} = \text{Pisa}} (\text{Studenti}))$

Nome	Matricola
Isaia	71523
Bonini	75649

Se volessi trovare il nome e l'anno di nascita degli studenti che hanno superato l'esame di BD con trenta, devo prima calcolarmi il prodotto tra le due tabelle *Studenti* × *Esami* e poi:

$$\pi_{\text{Nome}, \text{Nascita}} (\sigma_{\text{Materia} = \text{'BD'}} \wedge \text{Voto} = 30 \wedge \text{Matricola} = \text{Candidato} (\text{Studenti} \times \text{Esami}))$$

Intersezione

Laureati

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

Quadri

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

Laureati \cap Quadri

Matricola	Nome	Età
7432	Neri	54
9824	Verdi	45

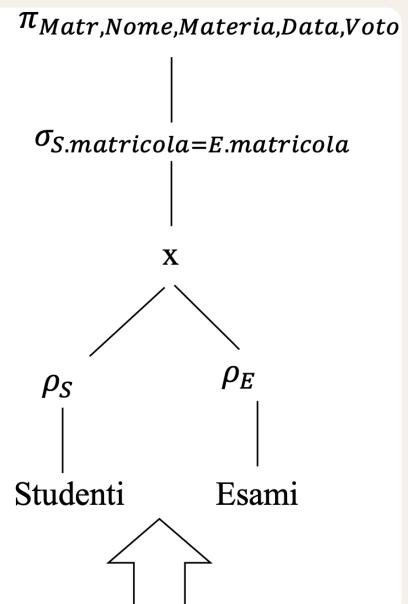
Vengono presi solamente i record che appartengono a entrambe le Tabelle.

Prodotto Cartesiano seguito da selezione

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

Esami

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



La composizione dei vari passaggi diventa:

$$\pi_{\text{Matr}, \text{Nome}, \text{Materia}, \text{Data}, \text{Voto}}(\sigma_{\text{matricola}=S.\text{matricola}}(E\text{sam}i \times \rho_S \text{ Studenti}))$$

Eseguo per prima cosa il prodotto cartesiano \times tra le due Tabelle e poi Selezione solamente i record che hanno il valore della {Matricola} uguale. Mi

interessa fare questo passaggio perchè devo controllare solamente i casi in cui le chiavi coincidono (altrimenti sarebbe inutile siccome il riferimento della FK è sbagliato). Proietto infine solamente le colonne con gli attributi che mi interessano.

Inner Join

Il prodotto cartesiano concatena le tuple non necessariamente correlate dal punto di vista semantico.

Impiegati

Impiegato	Reparto
Rossi	A
Neri	A
Neri	B

Reparti

Codice	Capo
A	Venere
B	Marte

Impiegati x Reparti

Impiegato	Reparto	Codice	Capo
Rossi	A	A	Venere
Neri	A	A	Venere
Neri	B	A	Venere
Rossi	A	B	Marte
Neri	A	B	Marte
Neri	B	B	Marte

Il prodotto cartesiano diventa utile quando è seguito da una selezione che mantiene solo le tuple con valori uguali sull'attributo chiave

La JOIN invece restituisce la tabella contenente le n-uple che hanno valori uguali per A_i e A_j dove R e S sono 2 tabelle di tipo diverso

$$R \bowtie_{Ai = Aj} S$$

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

Queste 2 scritture per indicare il prodotto cartesiano seguito da Selezione e la Theta-join sono equivalenti.

EX:

Join completa.

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

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

Join non completa

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

Reparto	Capo
B	Mori
C	Bruni

Impiegato	Reparto	Capo
Neri	B	Mori
Bianchi	B	Mori

Outer Join

Un Outer Join estende con valori NULL tutte le n-uple che verrebbero tagliate fuori con un Inner Join come nell'esempio sopra della join non completa.

Ne esistono di 3 tipi:

- *Left Join*: Mantiene tutte le n-uple del primo operando estendendo con NULL dove necessario
- *Right Join*: ... del secondo operando ...
- *Full Join*: ... di entrambi gli operandi ...

EX:

LEFT

Impiegati

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

Reparti

Reparto	Capo
B	Mori
C	Bruni

Impiegati \bowtie_{LEFT} Reparti

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

RIGHT

Impiegati

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

Reparti

Reparto	Capo
B	Mori
C	Bruni

Impiegati \bowtie_{RIGHT} Reparti

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

FULL

Impiegati

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

Reparti

Reparto	Capo
B	Mori
C	Bruni

Impiegati \bowtie_{FULL} Reparti

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

Groupby

$A_i \gamma f_i(R)$ dove A_i sono attributi di R e le f_i sono funzioni di aggregazione (min, max, count, ...)

Il valore del raggruppamento è una relazione calcolata come segue:

- Si partizionano le ennuple della relazione R , raggruppandole in base ai valori uguali degli attributi A_i .
- Per ogni gruppo, si calcolano le espressioni f_i .
- Per ogni gruppo, si restituisce una singola n-upla con:
 - Gli attributi A_i che identificano il gruppo.
 - I valori delle espressioni f_i calcolate.

EX.

$\{\text{Candidato}\} \gamma_{\{\text{count}(*), \text{min}(\text{Voto}), \text{max}(\text{Voto}), \text{avg}(\text{Voto})\}} (\text{Esami})$

Materia	Candidato	Voto	Docente		Materia	Candidato	Voto	Docente
DA	1	20	10	→	DA	1	20	10
LFC	2	30	20		MTI	1	24	30
MTI	1	30	30		LFC	2	30	20
LP	2	20	40		LP	2	20	40

Candidato	Count(*)	min(Voto)	max(Voto)	avg(Voto)
1	2	20	24	22
2	2	20	30	25

Trasformazioni algebriche

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

$$\sigma_{C_1}(\sigma_{C_2}(R)) \equiv \sigma_{C_1 \wedge C_2}(R)$$

$$\sigma_{C_1 \wedge C_2}(R \times S) \equiv \sigma_{C_1}(R) \times \sigma_{C_2}(S)$$

$$R \times (S \times T) \equiv (R \times S) \times T$$

$$(R \times S) \equiv (S \times R)$$

$$\sigma_C({}_X\gamma_F(R)) \equiv {}_X\gamma_F(\sigma_C(R))$$

Sono utili perchè consentono di diminuire il tempo di esecuzioni di azioni che hanno lo stesso risultato finale.

Proprietà

Idempotenza delle Proiezioni

$$\pi_X(E) = \pi_X(\pi_{XY}(E))$$

Una proiezione può essere trasformata in una cascata di proiezioni che eliminano i vari attributi in fasi diverse se E è definita su un insieme di attributi che contiene Y oltre che X. Non influisce sull'efficienza.

Anticipazione della selezione rispetto al Join

$$\sigma_F(E_1 \bowtie E_2) = E_1 \bowtie \sigma_F(E_2)$$

Se F fa riferimento solo ad attributi di E2. Aumenta l'efficienza della query perché la selezione riduce il numero delle righe di E2 prima del join.

DBMS

DBMS

Un DataBase Management System è un software progettato per gestire grandi quantità di dati in modo efficiente, assicurandone persistenza e condivisione. Grazie a meccanismi avanzati, il DBMS garantisce affidabilità, controllo degli accessi e gestione della concorrenza, prevenendo problemi di ridondanza e inconsistenza nei dati.

L'indipendenza fisica e logica del DBMS consente rispettivamente di separare l'organizzazione fisica dei dati dalla loro rappresentazione logica e di accedere ai dati senza considerare i dettagli del loro storage. Poiché i dati risiedono spesso su dischi, il trasferimento avviene in blocchi chiamati *pagine*, con dimensioni tipiche tra 4 e 64 KB. Un gestore di memoria permanente fornisce un'astrazione che nasconde le caratteristiche hardware e dei file, mentre un

gestore buffer ottimizza il trasferimento di pagine tra memoria permanente e temporanea.

Ogni pagina, contenente più record, è organizzata con un sistema di identificatori (*RID*) che permette di individuare rapidamente i dati con il *PID* (identificatore della pagina) e lo *slot* (posizione all'interno della pagina). Inoltre, il DBMS monitora modifiche e utilizzo delle pagine attraverso meccanismi come **pin count** e **dirty bit**, garantendo che le modifiche siano salvate correttamente per preservare l'affidabilità del sistema e dei dati.

Organizzazione della memoria

Noto il valore di una chiave, trovare il record di una tabella con idealmente un solo accesso al disco.

Metodo procedurale Hash file

In un hash file i record vengono allocati in una pagina il cui indirizzo dipende dal valore di chiave del record.

key → H(key) → page address

Una funzione hash comune è il resto della divisione intera e le collisioni vengono gestite con delle linked list.

Metodo tabellare

Si usa un indice, ovvero di un insieme ordinato di coppie $(k, r(k))$, dove

- k è un valore della chiave;
- $r(k)$ è un riferimento al record con chiave k .

L'indice è gestito di solito con un'opportuna struttura ad albero detta B^+ albero, la struttura più usata e ottimizzata dai DBMS.

Alberi di ricerca di ordine p

Un *albero di ricerca di ordine p* è una struttura dati utilizzata per organizzare i record, in cui ogni nodo contiene fino a $p - 1$ valori ordinati e fino a p puntatori a nodi figli.

L'albero deve rispettare due vincoli principali:

- i valori in ogni nodo devono essere ordinati
- i sottoalberi puntati dai nodi devono rispettare relazioni precise tra i valori di ricerca.

Questa struttura è utile per cercare e modificare i record memorizzati su disco, poiché ogni valore di ricerca può avere un puntatore associato al record corrispondente. Tuttavia, le operazioni di inserimento e cancellazione non garantiscono il bilanciamento dell'albero, rendendo a volte inefficiente la ricerca. Per risolvere questo problema, si utilizzano strutture bilanciate come i **B-tree** o i **B+-tree**, che assicurano una distribuzione uniforme dei nodi foglia sui livelli dell'albero.

B-tree

B⁺-tree

Indici

L'indice $(k, r(k))$ è una struttura che contiene informazioni sulla posizione di memorizzazione delle tuple sulla base del valore del campo chiave.

Esistono di 2 tipi:

- Indice statico: la struttura ad albero viene creata sulla base dei dati presenti nel DB
- Indice dinamico: la struttura ad albero viene aggiornata ad ogni operazione

Indice primario: Un indice primario utilizza la chiave primaria della tabella stessa, è un file ordinato i cui record sono di lunghezza fissa e sono costituiti da

due campi:

- Il primo campo è dello stesso tipo del campo chiave di ordinamento (primary key)
- Il secondo campo è un puntatore a un blocco del disco

Indice secondario: La chiave di ordinamento e la chiave di ricerca possono essere diverse, può essere definito su un campo non chiave che è una chiave candidata (valori univoci), o su un campo non chiave (valori duplicati)

- Il primo campo è dello stesso tipo del campo che non viene usato per ordinare il file ed è chiamato campo di indicizzazione
- Il secondo campo è un puntatore al blocco oppure un puntatore al record (RID)

- Tabella:

RID	Matr	Prov	An
1	106	MI	1972
2	102	PI	1970
3	107	PI	1971
4	104	FI	1968
5	100	MI	1970
6	103	PI	1972

- Indici

Matr	RID
100	5
102	2
103	6
104	4
106	1
107	3

Indice su Matr

An	RID
1968	4
1970	2
1970	5
1971	3
1972	1
1972	6

Indice su An

Esempi di indici per attributo chiave e non chiave

Ordinamento

L'algoritmo più comunemente utilizzato dai DBMS è il *Sort-Merge a Z vie*.
NP → numero pagine

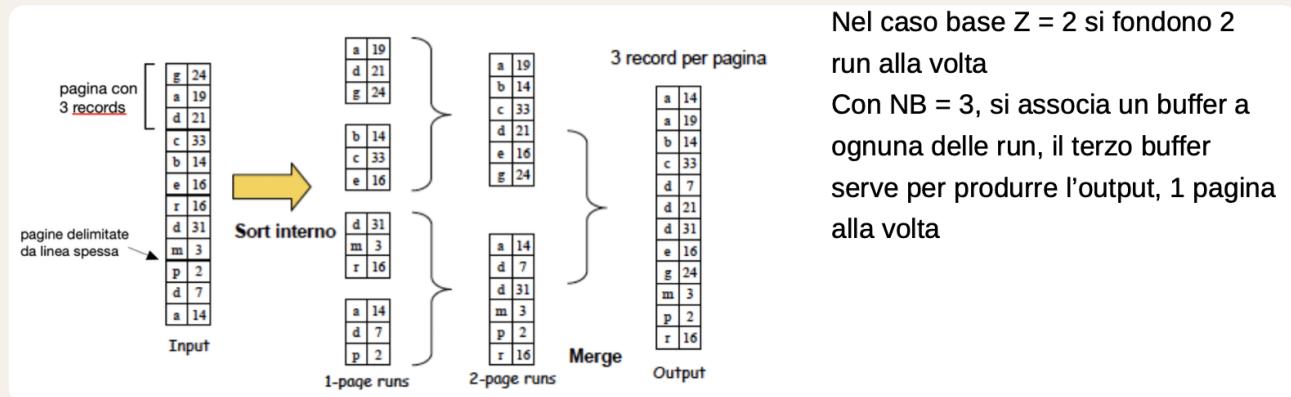
NB → numero buffer in memoria centrale (NM<NP)

L'algoritmo opera in 2 fasi:

- **Sort interno**: si leggono una alla volta le pagine del file; i record di ogni pagina vengono ordinati facendo uso di un algoritmo di sort interno (es. Quicksort); ogni pagina così ordinata, detta anche "run", viene scritta su disco in un file temporaneo
- **Merge**: operando uno o più passi di fusione, le "run" vengono fuse, fino a produrne una unica

EX.

NP = 4, NB = 3, Z = 2



EX2.

Supponiamo di avere un file di 1 GB e ogni pagina è di 1 MB. Ci saranno quindi 1024 pagine.

- Se abbiamo 5 buffer in memoria:
- Durante la fusione, possiamo combinare al massimo 4 run (1 buffer è riservato per scrivere l'output). Questo richiede più iterazioni per ridurre il numero totale di run.

Operatori Relazionali

Il Sort può essere utilizzato per:

- Query in cui compare **DISTINCT**, quindi per eliminare i duplicati
- Query che contengono GROUP BY

Proiezione

```
SELECT DISTINCT  
        Provincia  
FROM    Studenti R
```

1. Si legge R e si scrive T che contiene solo gli attributi della SELECT
2. Si ordinano su T tutti gli attributi
3. Si eliminano i duplicati

GROUP BY

Senza GROUP BY

- Si visitano i dati e si calcolano le funzioni di aggregazione.
- Select count(*) from Persone

Con GROUP BY

- Si ordinano i dati sugli attributi del GROUP BY, poi si visitano i dati e si calcolano le funzioni di aggregazione per ogni gruppo.
- Select cognome, count(cognome) from Persone group by cognome

Giunzione

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

Anche se logicamente il Join sia commutativo, dal punto di vista fisico vi è una chiara distinzione che influenzano le prestazioni:

- Operando sinistro → ESTERNO (R)
- Operando destro → INTERNO (S)

Il modo più banale per joinare due attributi delle relazioni è l'utilizzo un doppio for.

Nested Loops

Nel caso base in cui vi sia 1 buffer per R e 1 buffer per S, bisogna leggere 1 volta R e Nrec(R) volte S, ovvero tante volte quante sono le tuple della relazione esterna, per un totale di:

$$Npag(R) + Nrec(R) * Npag(S) \quad I/O$$

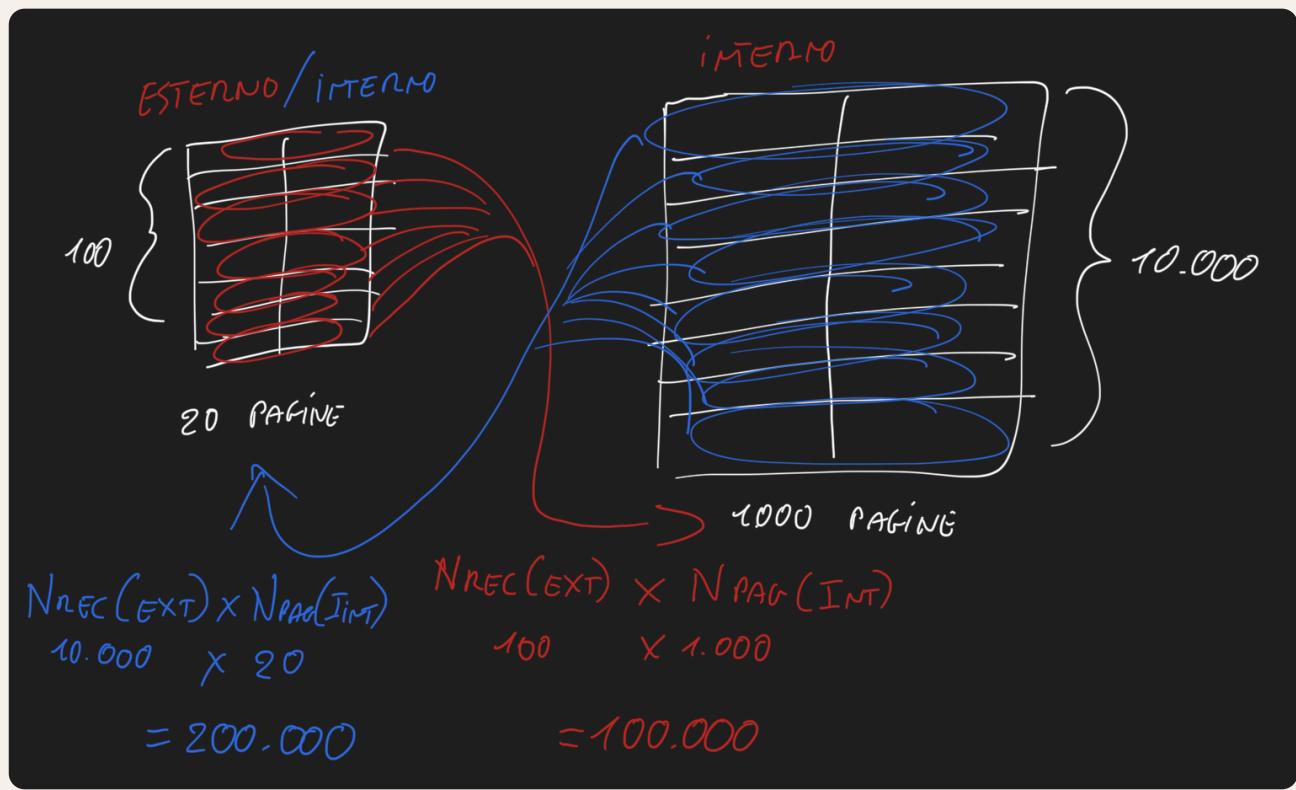
Se si fa il join tra Dipartimenti e Impiegati (DiplImp = NumDip) e si ha:

$$Npag(\text{Dipartimenti}) = 20 \quad Nrec(\text{Dipartimenti}) = 100$$

$$Npag(\text{Impiegati}) = 1000 \quad Nrec(\text{Impiegati}) = 10000$$

e considerando il caso base (1 buffer per ciascuna relazione):

- **Dipartimenti** esterna:
 - $Npag(\text{Dipartimenti}) + Nrec(\text{Dipartimenti}) * Npag(\text{Impiegati}) = 100.020$ I/O
- **Impiegati** esterna:
 - $Npag(\text{Impiegati}) + Nrec(\text{Impiegati}) * Npag(\text{Dipartimenti}) = 201.000$ I/O



Si sceglierà R come esterna e S come interna se:

$$Nrec(R) * Npag(S) < Nrec(S) * Npag(R)$$

che corrisponde a dire che le tuple di R sono più grandi di quelle di S

Page Nested Loop

L'idea principale è che, invece di esaminare ogni tupla individualmente, l'algoritmo carica intere pagine di una delle relazioni (tipicamente la relazione esterna) in memoria, e poi esegue il join per tutte le tuple all'interno di quella pagina con tutte le tuple della relazione interna.

Processo:

1. Carica una pagina della relazione esterna R in memoria.
2. Per ogni pagina di R, carica tutta la relazione interna S in memoria e per ogni tupla nella pagina di R, controlla il join con tutte le tuple di S.
3. Ripete il processo per tutte le pagine di R.

Costo notevolmente ridotto:

$$Npag(R) + Npag(R) * Npag(S) \quad I/O$$

Index Nested Loop

L'idea è la seguente:

1. *Scorrere tutte le tuple della relazione esterna R*
2. Per ogni tupla di R, **utilizzare l'indice sulla relazione interna S** per cercare le tuple corrispondenti. Questo permette di ridurre il numero di tuple da esplorare in S.

Per ogni tupla di R, si effettua una **ricerca dell'indice** nella relazione S per trovare tutte le tuple che soddisfano la condizione di join.

Costo:

$$N_{rec}(R) \times \text{costo di ricerca nell' indice di } S$$

Il costo di ricerca nell'indice sarà $O(\log N_{pag}(S))$ se l'indice è un B-Tree.

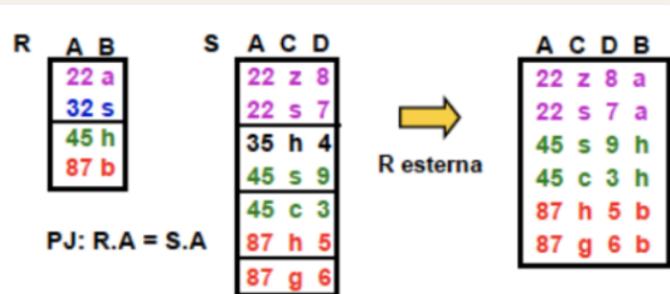
$$N_{rec}(R) \times O(\log N_{pag}(S))$$

Sort Merge

Si sfrutta il fatto che entrambi gli input sono ordinati per evitare di fare inutili confronti

SortMerge

```
r = first(R); s = first(S);
while r in R and s in S do
    if r.i = s.j
        avanza r ed s fino a che r.i ed s.j non
        cambiano entrambe, aggiungendo
        ciascun <r,s> al risultato
    else if r.i < s.j avanza r dentro R
    else if r.i > s.j avanza s dentro S
```



Piani di Accesso

```
// analisi lessicale e sintattica del comando SQL Q
SQLCommand parseTree = Parser.parseStatement(Q);

// analisi semantica del comando
Type type = parseTree.check();

// ottimizzazione dell'interrogazione
Value pianoDiAccesso = parseTree.Optimize();

// esecuzione del piano di accesso
pianoDiAccesso.open();
while !pianoDiAccesso.isDone() do
{
    Record rec = pianoDiAccesso.next();
    print(rec);
}
pianoDiAccesso.close();
```

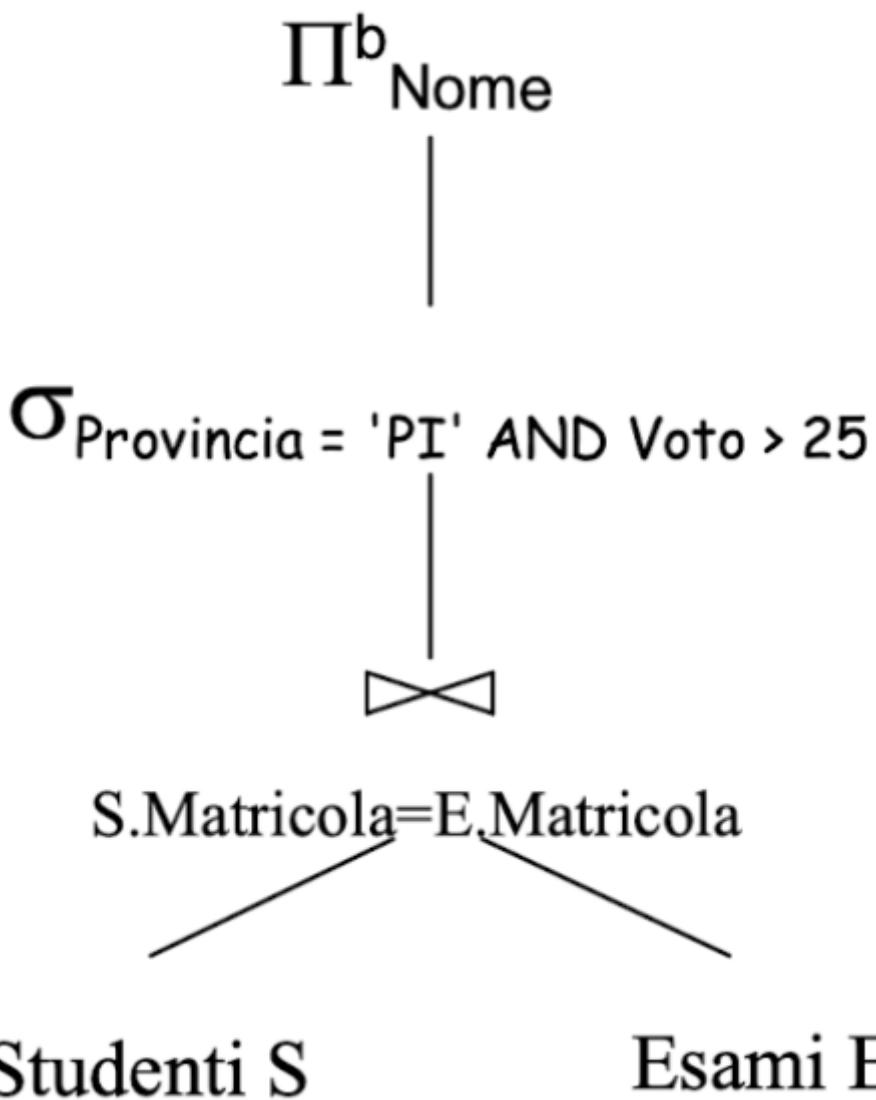
Bisogna scegliere il piano con costo minimo, fra possibili piani alternativi, usando le statistiche presenti nel catalogo.

EX.

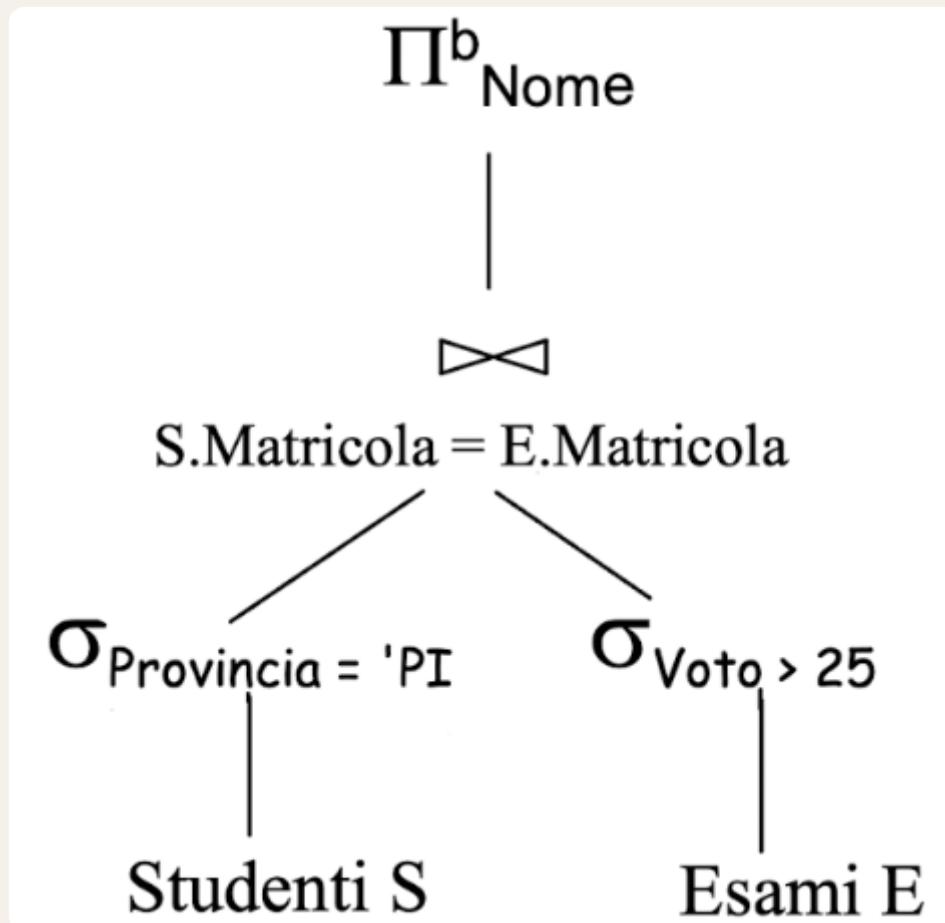
Ottimizzazione per questa query

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

Verifica la correttezza del comando, normalizzazione e semplificazione della condizione:



Trasformazione dell'albero con le regole di equivalenza:



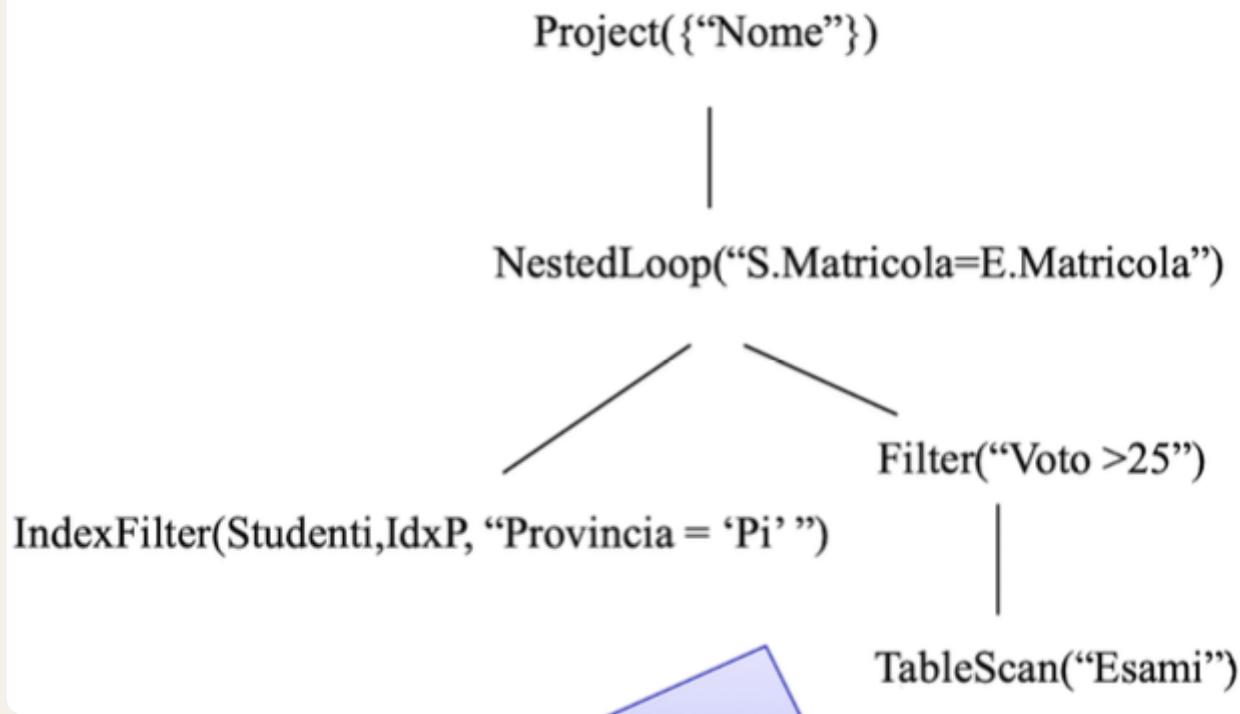
EX. di trasformazione

```
SELECT Matricola, Nome  
FROM Studenti  
Where Matricola IN (SELECT Matricola  
                      FROM Esami  
                      WHERE Materia = "BD")
```

⇒

```
SELECT Matricola, Nome  
FROM Studenti S, Esami E  
WHERE S.Matricola = E.Matricola AND Materia = "BD"
```

Il *piano di accesso* consiste nella scelta dell'algoritmo per implementare l'interrogazione che ci viene posta



Le foglie sono le tabelle e i nodi intermedi sono le modalità con cui gli accessi alle tabelle vengono effettuati.

Operatori fisici

Consideriamo i seguenti operatori: **Proiezione**, **Selezione**, **Raggruppamento**, **Join** che possono essere realizzati con algoritmi diversi, codificati in opportuni operatori fisici

Ad esempio **TableScan(R)** (immagine sopra) è l'operatore fisico per la scansione di R.

Ogni operatore fisico è un iteratore con metodi:

- **open** : *inizializza* lo stato dell'operatore, *alloca buffer* per gli input e l'output, *richiama ricorsivamente open sugli operatori figli*; viene anche **usato per passare argomenti** (ad es. la condizione che un operatore Filter deve applicare)
- **next** : usato per *richiedere un'altra tupla del risultato* dell'operatore; l'implementazione di questo metodo include *next sugli operatori figli* e codice specifico dell'operatore
- **isDone** : indica se vi sono *ancora valori da leggere*, in genere è booleano.

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

Operatori logici

Differenza tra R e O

Alcuni operatori accedono direttamente alla *relazione R*, altri prendono l'input dall'operatore `O` che sta sotto, il *figlio*

Operatore logico	Operatore fisico
<code>R</code>	<p><i>TableScan (R)</i> per la scansione di R;</p> <p><i>IndexScan (R, Idx)</i> per la scansione di R con l'indice Idx;</p> <p><i>SortScan (R, {A_i})</i> per la scansione di R ordinata sugli {A_i};</p>
$\pi^b_{\{A_i\}}$	<i>Project (O, {A_i})</i> per la proiezione dei record di O senza l'eliminazione dei duplicati;
$\pi_{\{A_i\}}$	<i>Distinct (O)</i> per eliminare i duplicati dei record ordinati di O;

	Filter (O, ψ) per la restrizione senza indici dei record di O ;
σ_{ψ}	IndexFilter (R, Idx, ψ) per la restrizione con indice dei record di R ;
$\tau_{\{A_i\}}$	Sort (O, $\{A_i\}$) per ordinare i record di O sugli $\{A_i\}$, per valori crescenti;

ψ è la condizione che deve essere rispettata.

	GroupBy (O, $\{A_i\}$, $\{f_i\}$) per raggruppare i record di O sugli $\{A_i\}$ usando le funzioni di aggregazione in $\{f_i\}$. <ul style="list-style-type: none">• Nell'insieme $\{f_i\}$ vi sono le funzioni di aggregazione presenti nella SELECT e nella HAVING.• L'operatore restituisce record con attributi gli $\{A_i\}$ e le funzioni in $\{f_i\}$.• I record di O sono ordinati sugli $\{A_i\}$;
--	--

	NestedLoop (O_E, O_I, ψ_J) per la giunzione con il nested loop e ψ_J la condizione di giunzione;
	PageNestedLoop (O_E, O_I, ψ_J) per la giunzione con il page nested loop;
\bowtie_{ψ_J}	IndexNestedLoop (O_E, O_I, ψ_J) per la giunzione con il index nested loop. L'operando interno O_I è un IndexFilter(R, Idx, ψ_J) oppure Filter (O, ψ') : con O un IndexFilter(R, Idx, ψ_J) : per ogni record r di O_E , la condizione ψ_J dell'IndexFilter è quella di giunzione con gli attributi di O_E sostituiti dai valori in r .
	SortMerge (O_E, O_I, ψ_J) per la giunzione con il sort-merge, con i record di O_E e O_I ordinati sugli attributi di giunzione.

EX.

```
SELECT *\nFROM R
```

R



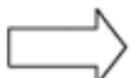
TableScan
(R)

Albero logico

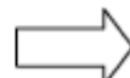
Piano di accesso

```
SELECT *\nFROM R\nGROUP BY A
```

$\tau_{\{A_i\}}$
|
R



SortScan
(R, {A})



Sort
({A})
|
TableScan
(R)

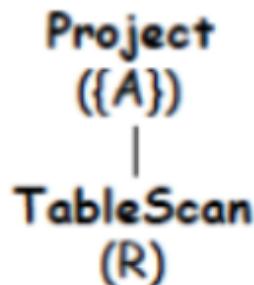
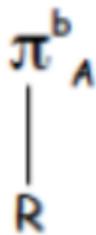
$\tau_{A_i} = \text{Sort}(O, \{A_i\})$, ma siccome il figlio O è R

⇒ Diventa un $\text{SortScan}(R, \{A_i\})$

⇒ Che equivale a $\text{TableScan}(R)$ seguito da $\text{Sort}(\{A_i\})$

Operatori fisici per la proiezione

**SELECT A
FROM R;**



Albero logico

Piano di accesso

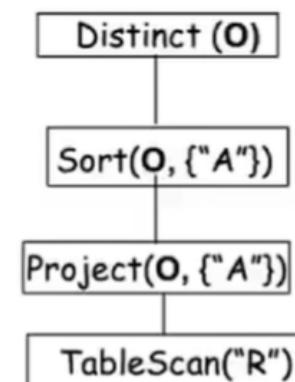
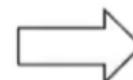
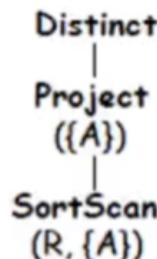
Si parte dalla radice dell'albero (Project), che rappresenta l'operatore chiamante. Questo esegue il metodo open, che a sua volta si propaga verso l'operatore TableScan, responsabile della scansione della tabella R. Conviene rimuovere i duplicati solo dopo aver selezionato le colonne di interesse, così da lavorare su meno dati e migliorare l'efficienza. ▲ Nel piano di esecuzione, è preferibile posizionare l'operatore Project prima di Sort, poiché questo riduce il numero di attributi da ordinare, rendendo l'operazione più veloce.

Dopo l'apertura, Project richiede a TableScan la prima tupla usando il metodo next, e successivamente le altre. TableScan legge la tupla corrispondente dalla relazione R e la invia a Project, che elimina le colonne non necessarie e restituisce il risultato.

**SELECT DISTINCT A
FROM R**

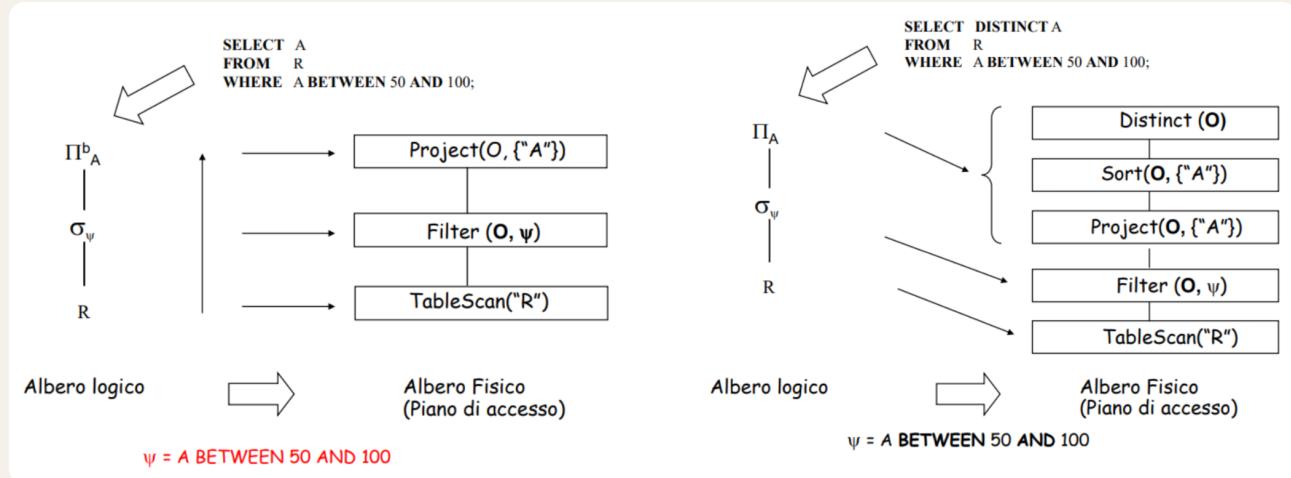


**SELECT DISTINCT A
FROM R;**

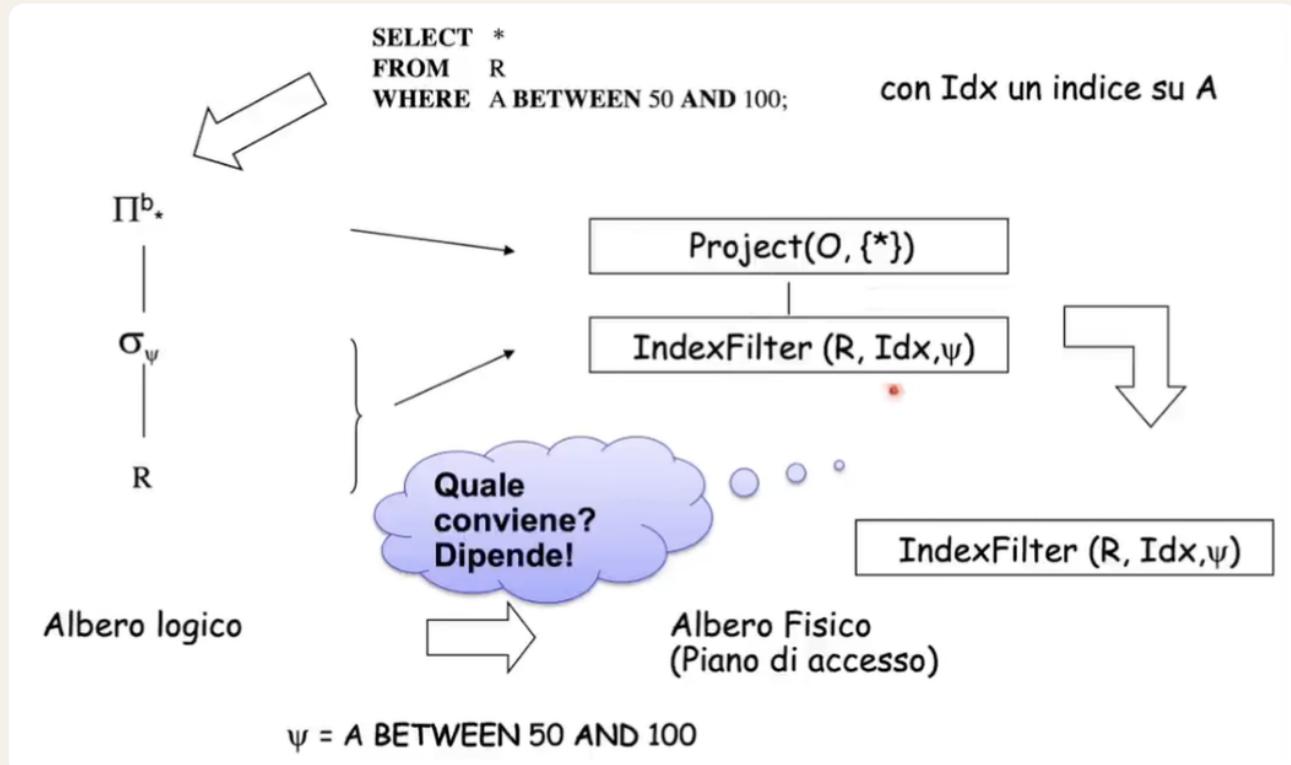


Conviene rimuovere i duplicati solo dopo aver selezionato le colonne di interesse, così da lavorare su meno dati e migliorare l'efficienza.
 Nel piano di esecuzione, è preferibile posizionare l'operatore Project prima di Sort, poiché questo riduce il numero di attributi da ordinare, rendendo l'operazione più veloce.

SENZA INDICE E DISTINCT



CON INDICE

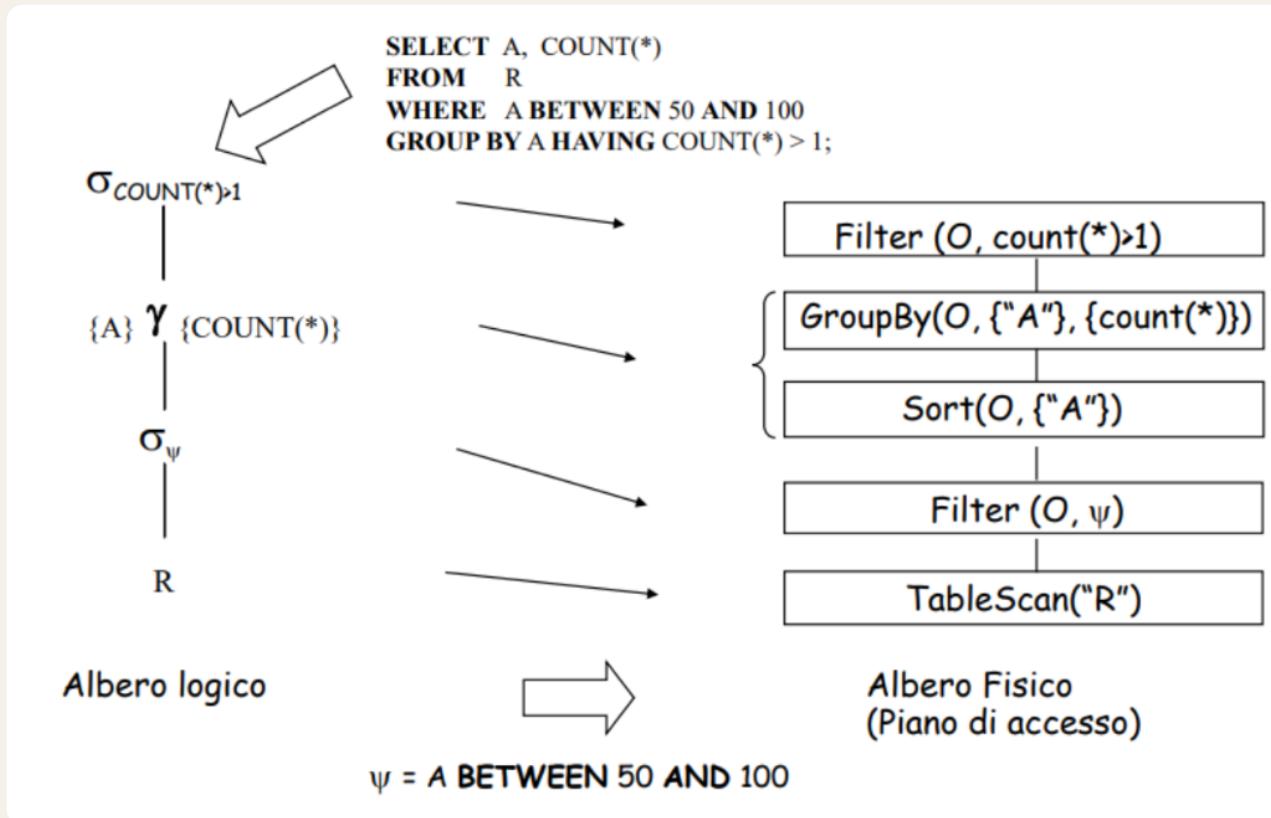


stesso esempio solo che posso sia usare:

- IndexFilter + Project

- solo IndexFilter tanto sto facendo una selezione con SELECT su tutti gli attributi IndexFilter = TableScan + Filter ma devo vedere caso per caso cosa conviene

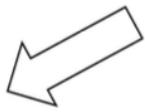
GROUP BY



il GROUP BY è quasi sempre seguito dal Sort a meno che non arrivino già ordinate le n-uple.

Solo dopo che ho fatto il conteggio e raggruppato, posso fare il filtraggio per l'HAVING, applicando direttamente sulle n-uple.

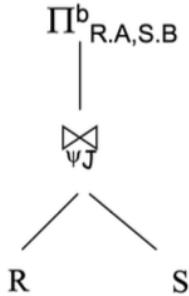
JOIN SENZA INDICE



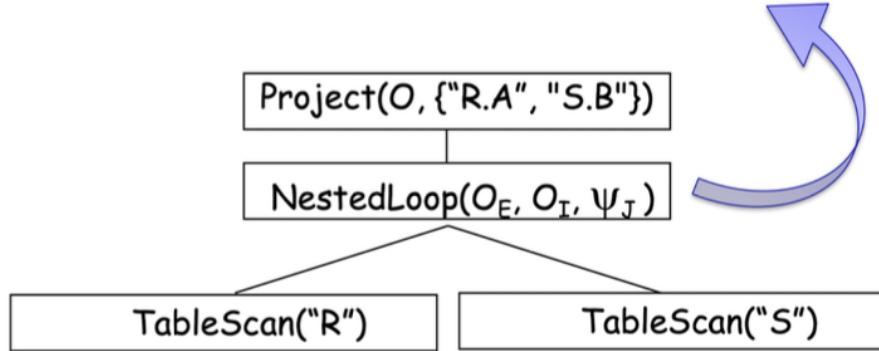
```

SELECT R.A, S.B
FROM   R , S
WHERE  R.A = S.B;
    
```

foreach record r in R do
 foreach record s in S do
 if $r_i = s_j$ then
 aggiungi $\langle r, s \rangle$ al risultato



Albero logico



Albero Fisico
(Piano di accesso)

$$\psi_J = R.A = S.B$$

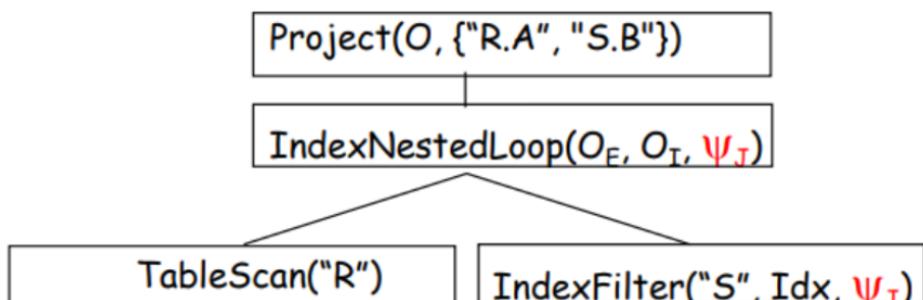
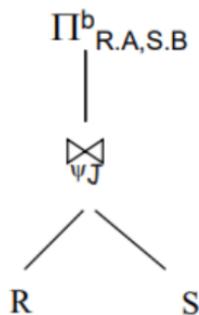
JOIN CON INDICE



```

SELECT R.A, S.B
FROM   R , S
WHERE  R.A = S.B;
    
```

con Idx un indice su $S.B$



Albero logico

Albero Fisico
(Piano di accesso)

$$\psi_J = R.A = S.B$$

Transizioni

Le transazioni rappresentano l'unità di lavoro elementare che modificano il contenuto di una base di dati.

Sono una sequenza di azioni di lettura e scrittura in memoria permanente e di elaborazioni di dati in memoria temporanea.

Proprietà ACID delle transazioni:

- **Atomicità**: La transazione deve essere eseguita con la regola del "tutto o niente".
- **Consistenza**: La transazione deve lasciare il DB in uno stato consistente, eventuali vincoli di integrità non devono essere violati.
- **Isolamento**: L'esecuzione di una transazione deve essere indipendente dalle altre.
- **Durability**: L'effetto di una transazione che ha fatto "commit work" non deve essere perso.

Sintatticamente un transazione è contornata dai comandi `begin transaction` ed `end transaction`; all'interno possono comparire i comandi di `commit work` e `rollback work`.

In un DBMS, le operazioni rilevanti sono la lettura (`ri[x]`) e la scrittura (`wi[x]`) di dati, considerati per semplicità come pagine. La lettura carica la pagina nel buffer se non già presente, mentre la scrittura modifica la pagina nel buffer, senza necessariamente salvarla subito su disco. Questo può causare perdite in caso di guasti.

Esistono 3 tipi di fallimenti:

1. **Fallimento di transazioni**: non comportano la perdita di dati in memoria temporanea né persistente
2. **Fallimenti di sistema**: comportano la perdita di dati in memoria temporanea ma non di dati in memoria persistente (es.: comportamento anomalo del sistema, caduta di corrente, guasti hardware sulla memoria centrale)
3. **Disastri**: comportano la perdita di dati in memoria persistente (es.: danneggiamento di periferica)

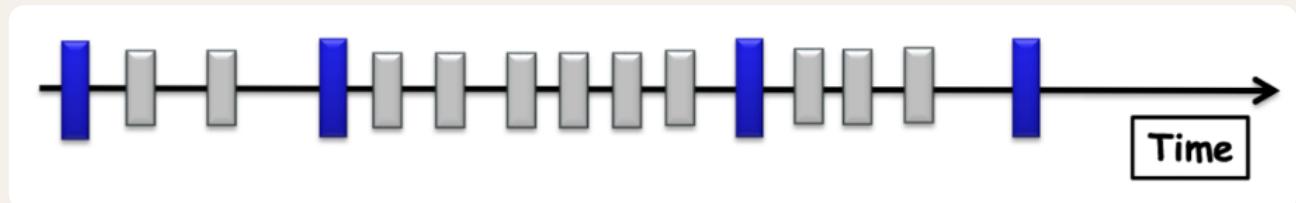
Gestore dell'affidabilità

Il gestore dell'affidabilità garantisce atomicità e persistenza delle transazioni. Gestisce comandi come `begin transaction`, `commit` e `rollback`, e ripristina il sistema dopo malfunzionamenti software (ripresa a caldo) o hardware (ripresa a freddo), utilizzando un log delle operazioni eseguite.

File di log

Il log è una struttura fisica che contiene tutte le informazioni e tutte le operazioni svolte dal DBMS e le scriviamo in un file detto ‘file di log’.

Il log contiene orario, data esecuzione transazione, il tipo di transazione che esegue le operazioni e che operazione viene eseguita.



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

Undo, Redo e Dump

- **B(T)** → record di **begin** relativo a T
- **C(T)** → record di **commit** relativo a T → *chiude in maniera definitiva la transazione*
- **A(T)** → record di **abort** relativo a T
- **U(T, O, BS, AS)** → record di **update** su oggetto O dove BS → before state, AS → after state
- **I(T, O, AS)** → record di **insert** su oggetto O dove AS → after state
- **D(T, O, BS)** → record di **delete** su oggetto O dove BS → before state

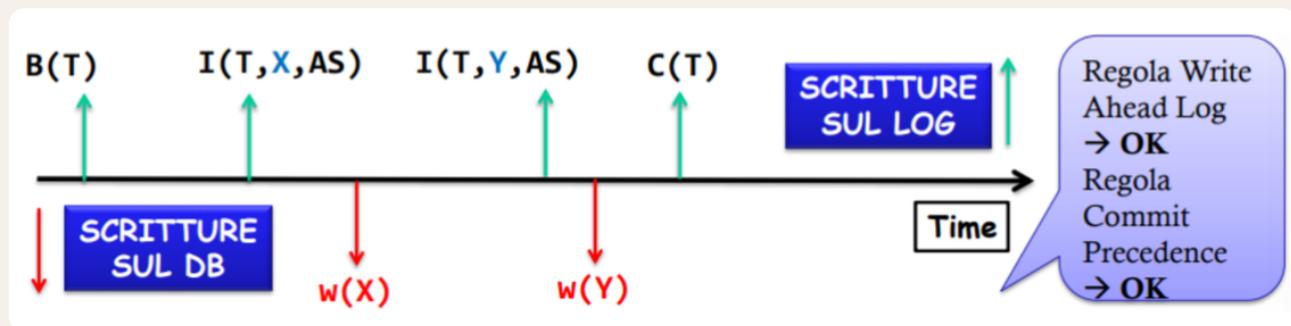
Il log registra le operazioni sulle transazioni per garantire atomicità (*undo*) e persistenza (*redo*). L'*undo* ripristina i dati precedenti, mentre il *redo* applica le modifiche. Il *dump* crea una copia completa della base di dati, eseguita in esclusiva quando il sistema è inattivo. Dopo il *dump*, un record nel log segnala il

completamento, permettendo di ripristinare il sistema combinando il backup e il log.

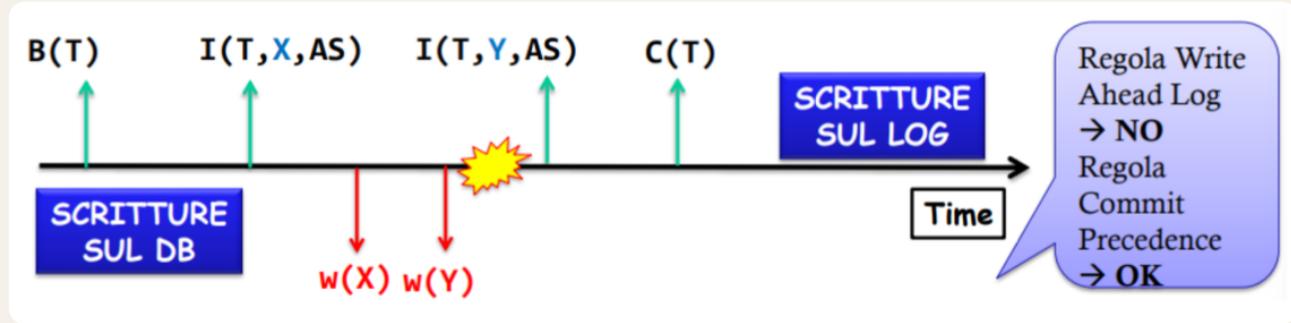
Regole di scrittura del log

Regola Write Ahead Log (WAL): la parte BS (before state) di ogni record di log deve essere scritta prima che la corrispondente operazione venga effettuata nella base di dati.

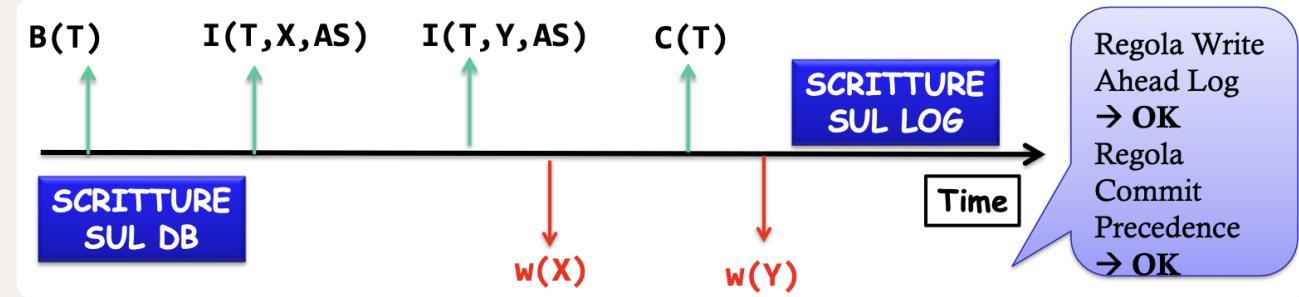
Regola di Commit Precedence: la parte AS (after state) di ogni record di log deve essere scritta nel log prima di effettuare il commit della transazione.



Questo esempio *va bene* e rispetta entrambe le regole spiegate sopra perché registra le operazioni di inserimento di X e di Y prima che effettivamente la relativa operazione di scrittura venga fatta sulla BD, inoltre le scrivo sul log entrambe prima di effettuare la commit $C(T)$.



Quest'altro esempio *non va bene* perché avviene una scrittura effettiva sulla BD prima che sia registrata la relativa operazione sul log, questo viola la regola WAL; Mentre l'altra è rispettata.



Questo va bene perchè nonostante la scrittura sul DB avvenga dopo, tutte le operazioni sono state registrate sul log prima del commit

Gestione delle modifiche alla BD:

- **Modifica libera:** Le modifiche possono essere scritte nel database stabile **prima** che la transazione termini (regola “disfare”).
- **Write Ahead Log (WAL):** Prima di aggiornare la BD, la vecchia versione deve essere salvata nel log per garantire che le modifiche possano essere annullate.

Gestione della terminazione:

- **Commit libero:** Una transazione può considerarsi completata **prima** che tutte le modifiche siano scritte nella BD stabile (regola “rifare”).
- **Commit Rule:** Le nuove versioni delle pagine devono essere registrate stabilmente nel log **prima** del commit per garantire la possibilità di ripetere le operazioni in caso di guasto.

Checkpoint

Il checkpoint è un punto di ripristino che indica che tutte le operazioni precedenti sono state salvate in modo permanente nel database. Serve per limitare il lavoro necessario durante il recupero da un guasto, poiché è sufficiente rifare solo le operazioni avvenute dopo l'ultimo checkpoint.

Come effettuare il checkpoint:

Metodo semplice

1. si sospende l'attivazione di nuove transazioni
2. si completano le precedenti, si allinea la base di dati
(ovvero si riportano su disco tutte le pagine "sporche" dei buffer)
3. si scrive nel log la marca CKP.
4. si riprende l'esecuzione delle operazioni.

Altro metodo

1. si scrive sul log una marca di inizio checkpoint che riporta l'elenco delle transazioni attive (`BeginCkp, {T1, ..., Tn}`)
2. il gestore del buffer riporta sul disco tutte le pagine modificate mentre le normali operazioni continuano
3. si scrive sul log una marca di `EndCkp`
4. la marca di EndCkp certifica che tutte le scritture avvenute prima del BeginCkp ora sono sul disco.

Le scritture avvenute tra BeginCkp e EndCkp forse sono sul disco e forse no.

Ripresa dai malfunzionamenti

- **Fallimenti di transazioni:** Si scrive sul log `T, abort` e si applica la procedura redo.
- **Fallimenti di sistema:** La BD viene ripristinata con il comando `Restart`, a partire dallo stato al checkpoint, facendo come segue:
 - Le T non terminate vanno disfatte
 - Le T terminate devono essere rifatte.
- **Disastri:** Si riporta in linea la copia più recente della BD e la si aggiorna rifacendo le modifiche delle T terminate normalmente.

Ripresa a caldo 🔥 :

Garantisce le proprietà di *atomicità* e *persistenza* delle transazioni dopo un guasto software. Si svolge in quattro fasi:

1. **Trova l'ultimo checkpoint:** Percorrere il log a ritroso fino all'ultimo checkpoint.
2. **Costruisci insieme UNDO e REDO:** Identificare le transazioni incomplete (UNDO) e completate (REDO).
3. **UNDO:** Percorrere il log all'indietro e annullare le operazioni delle transazioni in UNDO.

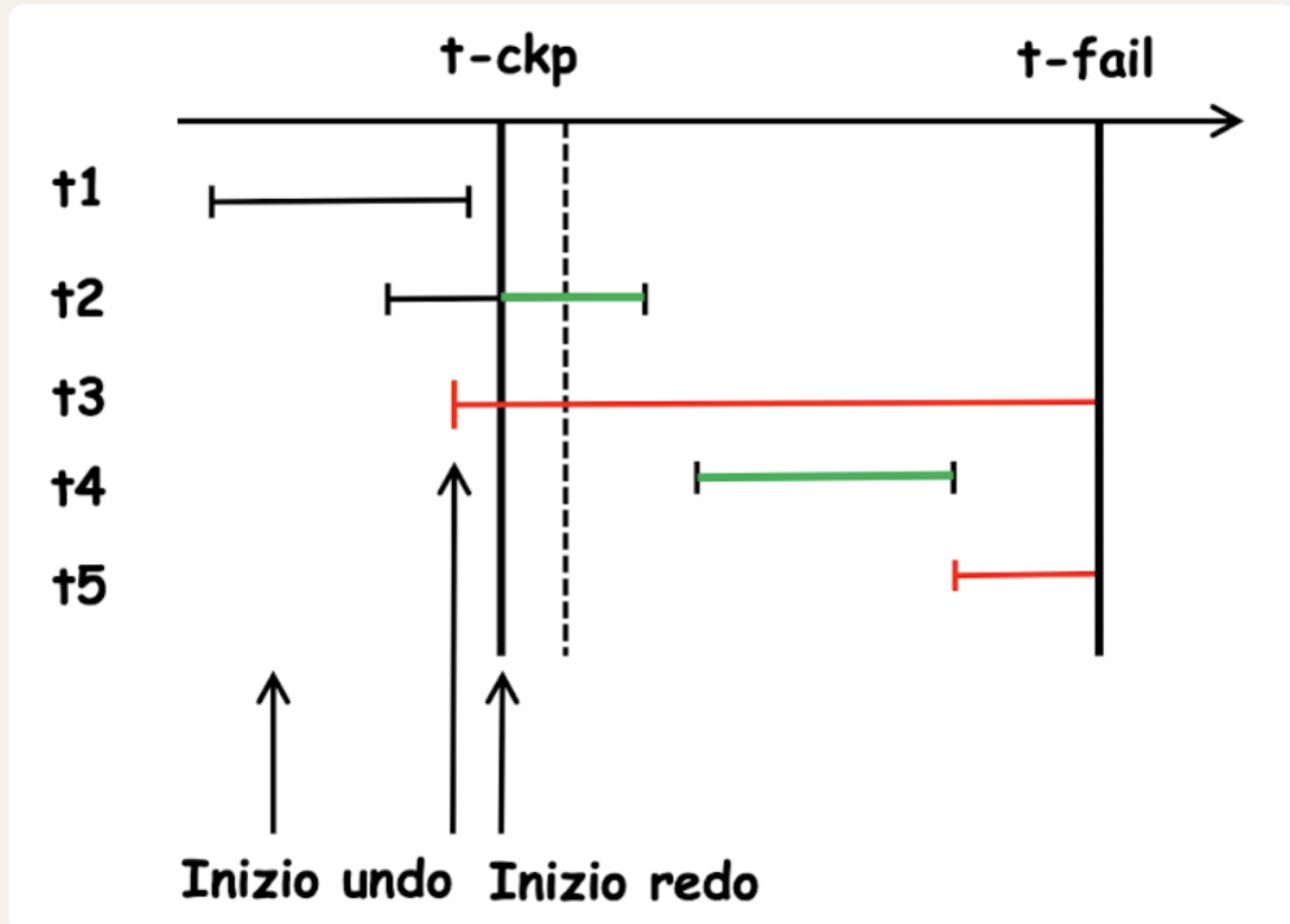
4. **REDO**: Percorrere il log in avanti e ripetere le operazioni delle transazioni in REDO.

Ripresa a freddo 🗃 :

Interviene in caso di guasto hardware o perdita di dati.

1. **Ripristino dal backup**: Si ripristina la base di dati dall'ultima copia disponibile.
2. **Ripristino dal log**: Eseguire le operazioni registrate nel log fino all'istante del guasto.
3. **Ripresa a caldo**: Completare la ripresa eseguendo il processo di ripresa a caldo.

EX.



Supponiamo di aver effettuato un checkpoint al tempo $t\text{-ckp}$ (che dura da linea spessa a linea tratteggiata e consideriamo come se quelle operazioni avvenissero dopo il ckp) e aver ricevuto un fallimento al tempo $t\text{-fail}$

Le linee indicano quando una determinata transazione tX effettua delle operazioni, delimitate da inizio e fine con due sbarre

- **T1:** Completata prima del checkpoint → Nessuna azione richiesta.
- **T2 e T4:**Terminate dopo il checkpoint → Devono essere rifatte.
- **T3 e T5:** Iniziate ma non terminate → Devono essere disfatte.

DDL

DDL

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

CREATE TABLE

Le tabelle vengono definite in SQL attraverso il comando **CREATE TABLE**:

- Definisce uno schema di relazione e ne crea un'istanza vuota.
- Specifica attributi, domini e vincoli.

```
CREATE TABLE <nome_tabella>
  (nome_colonna_1 tipo_colonna_1 clausola_default_1
  vincolo_di_colonna1,
   nome_colonna_2 tipo_colonna_2 clausola_default_2
  vincolo_di_colonna2,
   nome_colonna_3 tipo_colonna_3 clausola_default_3
  vincolo_di_colonna3,
   ...
  vincoli di tabella
)
```

```
CREATE TABLE IMPIEGATO (
    Matricola CHAR(6) PRIMARY KEY,
    Nome CHAR(20) NOT NULL,
    Cognome CHAR(20) NOT NULL,
    Dipart CHAR(15),
    Stipendio NUMERIC(9) DEFAULT 0,
    FOREIGN KEY(Dipart) REFERENCES Dipartimento(NomeDip),
    UNIQUE (Cognome,Nome)
)
```

Esempio creazione tabella impiegato

I tipi più comuni per i valori degli attributi sono:

- **CHAR(n)** → per stringhe di caratteri di lunghezza fissa n;
- **VARCHAR(n)** → per stringhe di caratteri di lunghezza variabile di al massimo n caratteri;
- **INTEGER** → per interi con la dimensione uguale alla parola di memoria standard dell'elaboratore;
- **REAL** → per numeri reali con dimensione uguale alla parola di memoria standard dell'elaboratore;
- **NUMBER(p,s)** → per numeri con p cifre, di cui s decimali;
- **FLOAT(p)** → per numeri binari in virgola mobile, con almeno p cifre significative;
- **DATE** → per valori che rappresentano istanti di tempo (in alcuni sistemi, come Oracle), oppure solo date (e quindi insieme ad un tipo TIME per indicare ora, minuti e secondi)

Modifica della tabella

ALTER TABLE : Permette di modificare la tabella con un insieme di comandi quali:

- **ADD [COLUMN]** : aggiunge una colonna

```
ALTER TABLE nome_tabella  
ADD [COLUMN] nome_col tipo_col default_col vincolo_col
```

- **DROP [COLUMN]** : elimina una colonna

```
ALTER TABLE nome_tabella  
DROP COLUMN nome_colonna {RESTRICT/CASCADE} #obbligatorio  
specificare
```

- **RESTRICT** : se un'altra tabella si ha un vincolo di integrità referenziale con questa colonna, l'esecuzione del comando DROP fallisce.
- **CASCADE** : eliminando la colonna, vengono eliminate tutte le dipendenze logiche di altre colonne dello schema da questa.

MODIFY : modifica una colonna

```
ALTER TABLE nome_tabella  
MODIFY nome_colonna tipo_col default_col vincoli_col
```

SET DEFAULT : Aggiunge l'assegnazione di valori di default

```
ALTER TABLE nome_tabella  
ALTER [COLUMN] nome_colonna  
SET DEFAULT valore_default
```

DROP DEFAULT : Elimina l'assegnazione di valori di default

```
ALTER TABLE nome_tabella  
ALTER [COLUMN] nome_colonna
```

DROP DEFAULT

ADD CONSTRAINT: Aggiunge vincoli di tabella

```
ALTER TABLE nome_tabella  
ADD CONSTRAINT nome_vincolo vincolo_di_tabella
```

DROP CONSTRAINT: Elimina vincoli di tabella

```
ALTER TABLE nome_tabella  
DROP CONSTRAINT nome_vincolo {RESTRICT/CASCADE} #obbligatorio
```

- **RESTRICT** non permette di eliminare vincoli di unicità e di chiave primaria su una colonna se esistono vincoli di chiave esterna che si riferiscono a tale colonna.
- **CASCADE** non opera questa restrizione.

DROP TABLE: Eliminare una tabella

Si può eliminare una tabella mediante l'istruzione DROP TABLE, nello standard SQL si possono anche specificare le opzioni RESTRICT / CASCADE:

- **RESTRICT** : se la tabella è utilizzata nella definizione di altri oggetti dello schema, la sua eliminazione viene impedita.
- **CASCADE** : vengono eliminate tutte le dipendenze degli altri oggetti dello schema da questa tabella

VINCOLI

I vincoli di integrità consentono di limitare i valori ammissibili per una determinata colonna della tabella in base a specifici criteri.

I vincoli di integrità *intra-relazionali* sono di 4 tipi:

- NOT NULL
- UNIQUE
- PRIMARY KEY
- CHECK

UNIQUE

Il vincolo UNIQUE utilizzato nella definizione dell'attributo indica che non ci possono essere due valori uguali in quella colonna.

Può anche essere riferito a coppie o insiemi di attributi. In quel caso assicura che la combinazione dei valori di quei campi sia unica, ma i singoli attributi non devono necessariamente esserlo da soli.

EX.

Abbiamo un vincolo `UNIQUE(nome, cognome)` sulla tabella Persone:

```
INSERT INTO Persone (nome, cognome) VALUES ('Mario', 'Rossi'); -- OK
INSERT INTO Persone (nome, cognome) VALUES ('Luigi', 'Rossi'); -- OK
INSERT INTO Persone (nome, cognome) VALUES ('Mario', 'Verdi'); -- OK
INSERT INTO Persone (nome, cognome) VALUES ('Mario', 'Rossi'); --
ERRORE
```

Non importa che "Mario" o "Rossi" siano ripetuti affinchè la n-upla "Mario Rossi" non venga inserita nuovamente.

PRIMARY KEY

Definisce la chiave primaria della tabella, ossia l'attributo che identifica univocamente un dato.

Implica sia il vincolo `UNIQUE` sia `NOT NULL` e può essere definito anche esso su un insieme di elementi.

CHECK

Richiede che una colonna o una combinazione di colonne soddisfi una condizione per ogni riga della tabella

EX.

Su una colonna

```
CREATE TABLE Dipendenti (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(50),
    età INT CHECK (età BETWEEN 18 AND 65)
)
```

Su più colonne

```
CREATE TABLE Prodotti (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(50),
    prezzo DECIMAL(10, 2),
    sconto DECIMAL(10, 2),
    CHECK (prezzo > 0 AND sconto ≤ prezzo)
)
```

FOREIGN KEY

Una FOREIGN KEY è un vincolo che collega una colonna (o un insieme di colonne) di una tabella a una colonna *chiave primaria* di un'altra tabella. Garantisce che i valori nella colonna della chiave esterna esistano nella tabella di riferimento oppure siano NULL.

REFERENCES

Il vincolo REFERENCES è utilizzato per specificare su quale colonna della tabella di destinazione si basa la chiave esterna.

```
CREATE TABLE Tabella1 (
    id INT PRIMARY KEY,
    nome VARCHAR(50)
```

```
)
```

```
CREATE TABLE Tabella2 (
    id INT PRIMARY KEY,
    id_tabella1 INT,
    FOREIGN KEY (id_tabella1) REFERENCES Tabella1(id)
)
```

Reazione alla violazione

Quando si crea un vincolo `FOREIGN KEY` in una tabella, in SQL si può specificare l'azione da intraprendere quando delle righe nella tabella riferita vengono cancellate o modificate.

ON DELETE :

- `NO ACTION` : Impedire il delete nella tabella riferita quando ci sono righe nella riferente che dipendono da essa.
- `CASCADE` : Generare un delete a catena sulla tabella riferente.
- `SET NULL` : Assegna NULL come valore alle righe che vengono cancellate nella riferente.
- `SET DEFAULT` : Assegna il valore di default quando la riga viene cancellata dalla tabella riferita.

ON UPDATE :

- `CASCADE` : I valori vengono aggiornati al valore della tabella riferita.
- `SET NULL` : I valori della tabella referente vengono impostati a NULL.
- `SET DEFAULT` : I valori della tabella referente vengono impostati a DEFAULT.
- `NO ACTION` : Rifiuta gli aggiornamenti che violino l'integrità referenziale.

VISTE

Una vista in SQL è una tabella virtuale che rappresenta il risultato di una query salvata. Non contiene dati propri, ma si basa sui dati presenti nelle tabelle sottostanti.

Le viste vengono utilizzate per semplificare l'accesso ai dati, migliorare la sicurezza (limitando l'accesso a specifiche colonne o righe) e creare rappresentazioni personalizzate dei dati per diversi utenti o applicazioni.

```
CREATE VIEW NomeVista [(ListaAttributi)]
AS SelectSQL [with [local | cascaded] check option]
```

I nomi delle colonne indicati nella lista attributi sono i nomi assegnati alle colonne della vista, che corrispondono ordinatamente alle colonne elencate nella select.

Se questi non sono specificati (infatti Lista Attributi è tra []), le colonne della vista assumono gli stessi nomi di quelli della/e tabella/e a cui si riferisce.

Creare una vista contenente la matricola, il nome, il cognome e lo stipendio degli impiegati del dipartimento di Amministrazione il cui stipendio è maggiore di 1000 euro

```
Create view ImpiegatiAmmin
(Matricola, Nome, Cognome, Stipendio) AS
Select Matr, Nome, Cognome, Stip
From Impiegato
Where Dipart = 'Amministrazione' and
Stipendio > 1000
```

Il contenuto della vista è dinamico ma la sua struttura non lo è:

- Se un dato viene aggiornato, il dato verrà visualizzato aggiornato anche sulla vista.
- Se viene aggiunta una colonna dalla tabella riferente, il cambiamento non avverrà sulla vista.

Viste di gruppo

Una vista di gruppo è una vista in cui una delle colonne è una funzione di gruppo. In questo caso è obbligatorio assegnare un nome alla colonna della vista corrispondente alla funzione di gruppo:

```
CREATE VIEW A3 (cod_fabbrica, numero_versioni) AS  
SELECT cod_fabbrica, sum(num_versioni)  
FROM Modelli  
GROUP BY cod_fabbrica  
  
CREATE VIEW A4 AS  
SELECT numero_versioni  
FROM A3  
  
# num_versioni → numero_versioni
```

Ovviamente si possono eliminare con questa sintassi:

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

RESTRICT: Cancella la view solo se non è riferita in altri oggetti

CASCADE: Cancella anche tutti i riferimenti

Nell'esempio di sopra, se faccio `DROP VIEW A3`:

- `CASCADE` eliminerebbe sia A3 che A4.
- `RESTRICT` invece non eliminerebbe A3 perché A4 è dipendente da essa.

Aggiornamento delle viste

Aggiornare le viste è utile in caso di accesso ai dati controllato! Magari uno non può aggiornare direttamente la tabella e passa dalla vista per aggiornarla.

Esempio:

`Impiegato(Nome, Cognome, Dipart, Ufficio, Stipendio)`

Il personale della segreteria non può accedere ai dati sullo stipendio ma può modificare gli altri campi della tabella, aggiungere e/o cancellare tuple

Si può controllare l'accesso tramite la definizione della VIEW:

```
CREATE VIEW Impiegato2 AS  
    SELECT Nome, Cognome, Dipart, Ufficio  
    FROM Impiegato  
    INSERT INTO Impiegato2 VALUES (...)
```

Una vista è aggiornabile se esiste una corrispondenza biunivoca tra le righe della vista e quelle della tabella sottostante.

Non è aggiornabile se usa `JOIN` o aggregazioni.

With Check Option

Assicura che le operazioni di inserimento e modifica effettuate tramite una vista rispettino la clausola `WHERE` definita nella vista stessa.

```
CREATE VIEW ImpiegatoRossi AS  
    SELECT *  
    FROM Impiegato  
    WHERE Cognome='Rossi'  
    WITH CHECK OPTION;
```

Solo gli inserimenti/modifiche con `Cognome='Rossi'` saranno accettati.

Operazioni con `Cognome='Bianchi'` falliranno.

- `CASCADED`: Verifica che le operazioni rispettino le condizioni della vista corrente e di tutte le viste da cui dipende.
- `LOCAL`: Verifica solo le condizioni della vista corrente e di quelle da cui dipende che usano a loro volta WITH CHECK OPTION.

Vantaggi delle viste

Semplificazione dei dati: Offrono una rappresentazione più intuitiva rispetto alla struttura normalizzata del database.

Sicurezza: Permettono di nascondere dati sensibili (es. password, stipendi) e limitano l'accesso degli utenti a specifiche parti del database.

Indipendenza logica: Consentono di mantenere applicazioni indipendenti dalla struttura logica del database.

Riduzione della duplicazione: Diverse viste sugli stessi dati evitano la replica delle informazioni.

PROCEDURE E TRIGGER

Procedure

Le procedure (o function) in SQL sono blocchi di codice salvati nel database che possono essere eseguiti per svolgere operazioni specifiche. Si usano per automatizzare processi ripetitivi o complessi.

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

Esempio di procedura che mi permette di contare gli studenti

Definizione: Si definiscono con CREATE PROCEDURE e si eseguono con il comando CALL.

Parametri: Possono accettare parametri in ingresso (IN), in uscita (OUT), o sia in ingresso che in uscita (INOUT).

Struttura: Contengono istruzioni SQL come SELECT, INSERT, UPDATE, DELETE, cicli (LOOP, WHILE), e condizioni (IF, CASE).

EX.

```
CREATE PROCEDURE AggiornaStipendio(IN id INT, IN incremento  
DECIMAL(10,2)  
BEGIN  
    UPDATE Impiegati  
    SET stipendio = stipendio + incremento  
    WHERE id_imp = id;  
END;  
  
CALL AggiornaStipendio(1, 500);
```

Trigger

I trigger in SQL sono programmi o eventi automatici che vengono eseguiti in risposta a determinati eventi su una tabella o una vista.

Eventi scatenanti: Si attivano su INSERT, UPDATE, o DELETE di una tabella.

Momento di esecuzione:

- **BEFORE** : Eseguito prima dell'evento.
- **AFTER** : Eseguito dopo l'evento.

Obiettivo: Controllare o verificare i dati, mantenere consistenza tra tabelle, generare log o notifiche.

I **trigger a livello di riga** vengono eseguiti una volta per ciascuna riga modificata durante una transazione. Si creano utilizzando la clausola **FOR EACH ROW** nell'istruzione **CREATE TRIGGER**.

I **trigger a livello di istruzione**, invece, si attivano una sola volta per ciascuna transazione, indipendentemente dal numero di righe modificate (ad esempio, se si inseriscono 100 righe, il trigger viene eseguito una sola volta).

```
CREATE TRIGGER ControlloStipendio
  BEFORE INSERT ON Impiegati
  DECLARE
    StipendioMedio FLOAT
  BEGIN
    SELECT avg(Stipendio) INTO StipendioMedio
      FROM Impiegati
     WHERE Dipartimento = :new.Dipartimento;
    IF :new.Stipendio > 2 * StipendioMedio
    THEN RAISE_APPL_ERR(-2061, 'Stipendio alto')
    END IF;
  END;
```

Esempio di trigger

I trigger possono essere classificati in base al loro comportamento:

- **trigger attivo** modifica lo stato della base di dati in risposta a determinati eventi.
- **trigger passivo** provoca il fallimento della transazione corrente se si verificano certe condizioni.
- **trigger INSTEAD OF**: specificano azioni da eseguire al posto di quelle che hanno attivato il trigger. Questi ultimi possono essere definiti solo su viste e devono operare a livello di riga. Ad esempio, un trigger INSTEAD OF può reindirizzare un'operazione **INSERT** verso una tabella diversa o aggiornare più tabelle sottostanti di una vista.

Controllo degli accessi

Ogni componente dello schema del database (tabelle, attributi, viste, domini, ecc.) può essere protetta. Il possessore della risorsa assegna privilegi agli altri utenti.

- **Amministratore** (sys/system): ha accesso completo.
- **Utenti normali**: possono ricevere privilegi specifici.

Ogni privilegio è caratterizzato da:

1. **Risorsa** a cui si riferisce.
2. **Azione permessa** (es. lettura, scrittura, modifica).
3. **Utente** che concede il privilegio.
4. **Utente** che riceve il privilegio.
5. **Trasmissibilità**: possibilità di trasferire il privilegio ad altri utenti.

Comando	Privilegio fornito
SELECT	Lettura di dati.
INSERT [Attributi]	Inserire record, specificando gli attributi interessati.
DELETE	Cancellare record.
UPDATE [Attributi]	Modificare record o solo gli attributi indicati.
REFERENCES [Attributi]	Definire chiavi esterne che riferiscono gli attributi.
WITH GRANT OPTION	Trasferire il privilegio ad altri utenti.

Concedere un privilegio:

```
GRANT <Privileges|ALL PRIVILEGES> ON Resource TO Users [WITH GRANT OPTION]
```

```
Grant INSERT, SELECT ON Esami TO Marco WITH GRANT OPTION
```

Revocare un privilegio:

```
REVOKE [GRANT OPTION FOR] Privileges ON Resource FROM Users [RESTRICT|CASCADE]
```

```
REVOKE INSERT, SELECT ON Esami FROM Marco CASCADE
```

Indici

Gli indici sono messi a disposizione dal dbms per rispondere alle query in maniera efficiente (hashtable, binary tree,...), trovando più velocemente quello che sto cercando tra le migliaia di dati nel mio database. Generalmente si fa sulle chiavi.

```
CREATE INDEX NomeIdx ON Tabella(Attributi)
CREATE INDEX NomeIdx ON Tabella WITH STRUCTURE = BTREE, KEY =
(Attributi)
DROP INDEX NomeIdx
```

Catalogo

Il catalogo è l'insieme di metadati dello schema del database. Include informazioni sulle tabelle e sui relativi attributi, come chiavi, vincoli e indici.

Consente la classificazione e gestione dei metadati.

I DBMS mantengono tabelle di sistema che registrano queste informazioni per supportare interrogazioni e aggiornamenti.

Normalizzazione

Normalizzazione

La normalizzazione è un procedimento volto all'eliminazione della ridondanza informativa e del rischio di incoerenza del database, è una procedura che permette di trasformare schemi non normalizzati in schemi che soddisfano una forma normale

Lo *schema di relazione universale* **U** di una base di dati è l'unione di tutti gli attributi di tutte le tabelle della base di dati

Una *forma normale* è una proprietà di una base di dati relazionale che ne garantisce la "qualità", cioè l'assenza di determinati difetti:

- Presenza di ridondanze

- Comportamenti poco desiderabili durante gli aggiornamenti

Comportamenti poco desiderabili → Anomalie

- *Ridondanza*: Valori ripetuti nelle n-uple
- *Anomalia di aggiornamento*: Se un valore viene modificato, allora devo modificarlo per tutte le sue ripetizioni manualmente
- *Anomalia di cancellazione*: Stesso problema dell'aggiornamento
- *Anomalia di inserimento*: Spiego con un esempio che è più chiaro

Matricola	NomeStudente	Corso	NomeDocente
12345	Mario Rossi	Matematica	Prof. Verdi
67890	Anna Bianchi	Informatica	Prof. Neri
.			

- Se vogliamo inserire un nuovo corso, ad esempio *Fisica* insegnato dal *Prof. Bianchi*, **senza avere ancora studenti iscritti**, non possiamo inserire la riga nella tabella, perché il valore della colonna Matricola e NomeStudente sarebbe NULL, il che potrebbe non essere permesso a causa di vincoli di integrità.

Linee guida

- Si progetti ogni schema relazionale in modo tale che sia **semplice spiegarne il significato**
- Si **eviti di porre in relazione di base attributi i cui valori possono essere frequentemente nulli**. Se è inevitabile, ci si assicuri che essi si presentino solo in casi eccezionali e che non riguardino una maggioranza di tuple nella relazione
- Si progettino gli schemi relazionali in modo che nelle relazioni **non siano presenti anomalie di inserimento, cancellazione o modifica**. Se sono presenti e le voglio mantenere, le si rilevi chiaramente e ci si assicuri che i programmi che aggiornano la base di dati operino correttamente
- Si progettino schemi di relazione in modo tale che essi **possano essere riuniti tramite JOIN con condizioni di uguaglianza su attributi che sono o chiavi primarie o chiavi esterne** in modo da garantire che non vengano generate tuple spurie.

Dipendenza funzionali

Istanza valida di R → un insieme di tuple che soddisfa tutti i vincoli definiti sullo schema della relazione R

Sia r una istanza valida su $R(T)$, siano X e Y due sottoinsiemi di attributi non vuoti di T:

- Allora esiste in r una dipendenza funzionale da X a Y se, per ogni coppia di n-uple t_1 e t_2 di r con gli stessi valori su X , risulta che t_1 e t_2 hanno gli stessi valori anche su Y

EX.

StudentID	Nome	Corso	Voto
12345	Mario Rossi	Matematica	30
67890	Anna Bianchi	Informatica	28
12345	Mario Rossi	Informatica	27
67890	Anna Bianchi	Matematica	29

$R(T)$ dove $T = \{StudentID, Nome, Corso, Voto\}$

$X = \{StudentID\}$

$Y = \{Nome\}$

$X \rightarrow Y$ X determina funzionalmente Y o Y è determinato da X se:

$\forall r$ istanza valida di R,

$\forall t_1, t_2 \in r \mid t_1[X] = t_2[X]$ allora $t_1[Y] = t_2[Y]$

$t_1[X] = 67890$ $t_2[X] = 67890$

$t_1[Y] = AnnaBianchi$ $t_2[Y] = AnnaBianchi$

Y è determinato funzionalmente da X

Il concetto di Dipendenza Funzionale generalizza il concetto di chiave

La chiave è sempre una dipendenza funzionale con tutti gli attributi a destra

Dipendenze Funzionali Atomiche

Una dipendenza funzionale atomica è una DF dove a destra ho solo un Attributo

DF:

{ CodiceLibro → Titolo

NomeNegozio → IndNegozio

CodiceLibro, NomeNegozio → IndNegozio, Titolo, Quantità }

Abbiamo delle ridondanze perché per conoscere il Titolo non ho bisogno del NomeNegozio ma solo del CodiceLibro.

Possiamo scomporre quindi la terza DF in 3 dipendenze atomiche:

1. CodiceLibro, NomeNegozio → IndNegozio
2. CodiceLibro, NomeNegozio → Titolo
3. CodiceLibro, NomeNegozio → Quantità

Da qua possiamo vedere bene che 1 e 2 sono ridondanti e quindi inutili.

Ogni DF $X \rightarrow A_1, A_2, A_3, \dots$

Può essere scomposta in:

$X \rightarrow A_1, X \rightarrow A_2, X \rightarrow A_3, X \rightarrow \dots$

{ CodiceLibro → Titolo

NomeNegozio → IndNegozio

CodiceLibro, NomeNegozio → Quantità }

$R < T, F >$ denota uno schema con Attributi T e DF F

Una DF è *Completa* quando $X \rightarrow Y$ e $\forall W \subseteq X$ non vale $W \rightarrow Y$, ovvero **minimale**.

Se provo a togliere attributi da X allora non ottengo la stessa DF

Se X è *superchiave*, allora X determina ogni altro attributo di $X \rightarrow T$ (potrebbe essere non minimale, ex. $X = \{\text{StudentID}, \text{Nome}\}$).

Se X è *chiave*, allora $X \rightarrow T$ è una DF **completa**.

Dipendenze implicate

sia F un insieme di DF sullo schema R , diremo che F implica logicamente $X \rightarrow Y$ ($F \models X \rightarrow Y$), se ogni istanza r di R che soddisfa F soddisfa anche $X \rightarrow Y$.

EX.

$R < T, F >$ con $F = \{X \rightarrow Y, X \rightarrow Z\}$ e $X, Y, Z \subseteq T$

$$\begin{aligned} t_1[X] = t_2[X] &\Rightarrow t_1[Y] = t_2[Y] \\ t_1[X] = t_2[X] &\Rightarrow t_1[Z] = t_2[Z] \\ t_1[X] = t_2[X] &\Rightarrow t_1[YZ] = t_2[YZ] \end{aligned}$$

Pertanto $\{X \rightarrow Y, X \rightarrow Z\} \models X \rightarrow YZ$

$R < T, F >$ con $F = \{X \rightarrow Y, Y \rightarrow Z\}$ e $X, Y, Z \subseteq T$

$\Rightarrow \{X \rightarrow Y, Y \rightarrow Z\} \models X \rightarrow Z$

Assiomi di Armstrong

- Se $Y \subseteq X$, allora $X \rightarrow Y$ (Riflessività)
- Se $X \rightarrow Y$, $Z \subseteq T$, allora $XZ \rightarrow YZ$ (Arricchimento)
- Se $X \rightarrow Y$, $Y \rightarrow Z$, allora $X \rightarrow Z$ (Transitività)

Derivazioni

$Z \subseteq Y, \{X \rightarrow Y\} \vdash X \rightarrow Z$ (Decomposizione)

$\{X \rightarrow Y, X \rightarrow Z\} \vdash X \rightarrow YZ$ (Unione)

Dimostrazione Unione:

1. $X \rightarrow Y$ (per ipotesi)
2. $X \rightarrow XY$ (per arricchimento con X da 1)
3. $X \rightarrow Z$ (per ipotesi)
4. $XY \rightarrow YZ$ (per arricchimento con Y da 3)
5. $X \rightarrow YZ$ (per transitività da 2, 4)

Correttezza e Completezza degli assiomi di Armstrong

Attraverso gli assiomi di Armstrong, si può mostrare l'equivalenza della nozione di implicazione logica ($|=$) e di quella di derivazione ($|-$):

- Se una dipendenza è derivabile con gli assiomi di Armstrong allora è anche implicata logicamente (correttezza degli assiomi)
- Se una dipendenza è implicata logicamente allora è anche derivabile dagli assiomi (completezza degli assiomi)

implicazione logica $|=$

Una dipendenza $X \rightarrow Y$ è **implicata logicamente** dall'insieme F di dipendenze se, in ogni situazione possibile dove F è vero, anche $X \rightarrow Y$ risulta vero.

Ad esempio, se conosciamo queste dipendenze:

- $A \rightarrow B$,
- $B \rightarrow C$,

allora possiamo dire che è **implicato logicamente** che $A \rightarrow C$. Questo perché, se $A \rightarrow B$ e $B \rightarrow C$ sono vere, automaticamente $A \rightarrow C$ sarà vero.

derivazione |-

La derivazione invece è come un **processo meccanico**: partendo dalle dipendenze F (che sappiamo essere vere), possiamo applicare gli **assiomi di Armstrong** per arrivare a $X \rightarrow Y$.

Ad esempio:

1. Da $A \rightarrow B$ e $B \rightarrow C$, possiamo usare l'**assioma di transitività** per concludere che $A \rightarrow C$.
2. Quindi, $A \rightarrow C$ è **derivabile** dagli assiomi di Armstrong.

Chiusura di un insieme F

La **chiusura** di un insieme F di dipendenze funzionali, indicata con F^+ , è **l'insieme di tutte le dipendenze funzionali** che possono essere **derivate logicamente** da F , utilizzando gli **assiomi di Armstrong**.

Algoritmo per calcolare X_F^+

Sia X un insieme di Attributi e F un insieme di Dipendenze, $X_F^+?$

1. $X^+ = X$
2. Se fra le dipendenze di F c'è $Y \rightarrow A$ con $Y \subseteq X^+$, allora si inserisce A in X^+
3. Si ripete 2) fino a quando non ci sono più attributi da aggiungere
4. Si da in output $X_F^+ = X^+$

EX.

Vogliamo trovare gli attributi che sono determinati funzionalmente da un insieme di dipendenze A e D

$F = \{DB \rightarrow E, B \rightarrow C, A \rightarrow B\}$. **Trovare** AD^+

1. $X^+ = AD$
 - Siccome F contiene $A \rightarrow B$, aggiungiamo B
2. $X^+ = ADB$
 - Siccome F contiene $DB \rightarrow E$, aggiungiamo E

3. $X^+ = ADBE$

- Siccome F contiene $B \rightarrow C$, aggiungiamo C

4. $X^+ = ADBEC$

Chiavi e Attributi primi

Dato lo schema R $\langle T, F \rangle$, diremo che un insieme di attributi $W \rightarrow T$ è una chiave candidata di R, se rispetta queste condizioni:

- $W \rightarrow T \in F^+$. (W superchiave)
- $\forall V \subseteq W, V \rightarrow T \notin F^+$. (se $V \in W, V$ non superchiave)

Attributo primo: attributo che appartiene almeno a una chiave

- $F = \{DB \rightarrow E, B \rightarrow C, A \rightarrow B\}$, trovare $(AD)^+$:

$$X^+ = AD$$

$$X^+ = ADB$$

$$X^+ = ADBE$$

$$X^+ = ADBEC$$

- **AD** è superchiave?

- Si poiché contiene tutti gli attributi

- **A** è superchiave?

- $A \rightarrow B, A \rightarrow BC$, si ferma \rightarrow non è superchiave

- **ABD** è superchiave?

- $(ABD)^+$ è analoga a $(AD)^+$, perché ABD è più grande di AD, quindi è superchiave

- **ABC** è superchiave?

- ABC stesso, quindi non è superchiave

Se con l'algoritmo mi fermo prima di ottenere tutti gli attributi nel mio X^+ , allora non è superchiave

<u>Impiegato</u>	<u>Stipendio</u>	<u>Progetto</u>	<u>Bilancio</u>	<u>Funzione</u>
------------------	------------------	-----------------	-----------------	-----------------

$F = \{ \text{Impiegato} \rightarrow \text{Stipendio}, \text{Progetto} \rightarrow \text{Bilancio},$
 $\text{Impiegato Progetto} \rightarrow \text{Funzione} \}$

$\{\text{Impiegato}\}^+ = \{\text{Impiegato}, \text{Stipendio}\}$

$\{\text{Progetto}\}^+ = \{\text{Progetto}, \text{Bilancio}\}$

$\{\text{Impiegato}, \text{Progetto}\}^+ = \{\text{Impiegato}, \text{Progetto}, \text{Stipendio}, \text{Bilancio}, \text{Funzione}\}$

- $\{\text{Impiegato}\}$ **non** è superchiave
- $\{\text{Progetto}\}$ **non** è superchiave
- $\{\text{Impiegato}, \text{Progetto}\}$ **è** superchiave

L'algoritmo per trovare tutte le chiavi si basa su due proprietà:

1. Se un attributo A di T *non appare a destra* di alcuna dipendenza in F, allora **A appartiene ad ogni chiave di R**
2. Se un attributo A di T *appare a destra di qualche dipendenza* in F, ma *non appare a sinistra di alcuna dipendenza* (non banale), allora A **non appartiene ad alcuna chiave** perché da quell'attributo non potrei derivare nulla.

EX.

$$T = (A, B, C, D, E, F)$$

$$F = \{C \rightarrow D, CF \rightarrow B, D \rightarrow C, F \rightarrow E\}$$

Come minimo la chiave contiene A e F perché non compaiono mai a destra di nessuna dipendenza.

E non farà parte della chiave perché non compare mai a sinistra ma compare a destra

$$AF^+ = AFE, \text{ mancano BCD} \rightarrow \text{no superchiave}$$

A questo punto provo tutte le altre combinazioni e vedo se includo tutti gli attributi

$AFB^+ = AFBE \rightarrow$ no superchiave

$AFC^+ = AFCDBE \rightarrow$ superchiave

$AFD^+ = AFDCBE \rightarrow$ superchiave

Copertura Canonica

Se 2 insiemi DF, F e G sono equivalenti $F \equiv G$ se e solo se $F^+ = G^+$

Se $F \equiv G$, allora F è copertura di G e viceversa

Attributo Estraneo

A è un attributo estraneo in $X \rightarrow Y$ se e solo se $X - \{A\} \rightarrow Y$

Esempio:

Orari(CodAula, NomeAula, Piano, Posti, Materia, CDL, Docente, Giorno, Ora)

• se vale

- Docente, Giorno, Ora \rightarrow CodAula
- Docente, Giorno \rightarrow Ora

• allora

- Docente, Giorno \rightarrow CodAula
- (quindi) nella prima dipendenza Ora è attributo estraneo

Dipendenza Ridondante

F è una dipendenza ridondante se e solo se $(F - \{X \rightarrow Y\})^+ \rightarrow F^+$

Esempio:

Orari(CodAula, NomeAula, Piano, Posti, Materia, CDL, Docente, Giorno, Ora)

• se vale

- Docente, Giorno, Ora \rightarrow CodAula
- CodAula \rightarrow NomeAula

• è inutile avere anche

- Docente, Giorno, Ora \rightarrow NomeAula

EX.

$$F_3 = \{A \rightarrow B, A \rightarrow C\}$$

- F_3 non presenta attributi estranei

$$F_2 = \{A \rightarrow B, AB \rightarrow C\}$$

- F_2 non è ridondante ma presenta un attributo estraneo, perché B può essere eliminato dal primo membro della seconda dipendenza ($A^+ = A$; $A^+ = AB$; $A^+ = ABC$) (quindi equivale a F_3)

$$F_1 = \{A \rightarrow B, AB \rightarrow C, A \rightarrow C\}$$

- F_1 è ridondante, perché $\{A \rightarrow B, AB \rightarrow C\}$ implica $A \rightarrow C$ (quindi equivale a F_2)

F è detta una *copertura canonica* se e solo se:

- la parte destra di ogni DF in F è UN attributo (non 2, non 3, 1 e solo 1)
- non esistono attributi estranei
- nessuna dipendenza in F è ridondante

Algoritmo Copertura Canonica

Esempio ESISTENZA DELLA COPERTURA CANONICA

Impiegato (Matricola, Cognome, Grado, Retribuzione, Dipartimento, Supervisore, Progetto, Anzianità)

Consideriamo il seguente insieme di dipendenze funzionali

$\{M \rightarrow RSDG, MS \rightarrow CD, G \rightarrow R, D \rightarrow S, S \rightarrow D, MPD \rightarrow AM\}$

$\{M \rightarrow RSDG, MS \rightarrow CD, G \rightarrow R, D \rightarrow S, S \rightarrow D, MPD \rightarrow AM\}$

1. Creiamo le dipendenze funzionali atomiche

$\{M \rightarrow R, M \rightarrow S, M \rightarrow D, M \rightarrow G, MS \rightarrow C, MS \rightarrow D, G \rightarrow R,$
 $D \rightarrow S, S \rightarrow D, MPD \rightarrow A, MPD \rightarrow M\}$

2. Eliminare gli attributi estranei:

- è possibile eliminare S dal primo membro di $MS \rightarrow C$ e $MS \rightarrow D$ perché $M \rightarrow S$ (si ottiene da $M \rightarrow D$ e $D \rightarrow S$)
- È possibile eliminare D dal primo membro di $MPD \rightarrow A$ e $MPD \rightarrow M$ poiché $M \rightarrow D$

$\{M \rightarrow R, M \rightarrow S, M \rightarrow D, M \rightarrow G, M \rightarrow C, M \rightarrow D, G \rightarrow R,$
 $D \rightarrow S, S \rightarrow D, MP \rightarrow A, MP \rightarrow M\}$

3. Si trova l'insieme di dipendenza funzionali non ridondante: eliminiamo le dipendenze ottenibili da altre:

$M \rightarrow R$ (deriva da $M \rightarrow G$ e $G \rightarrow R$)

$M \rightarrow S$ (deriva da $M \rightarrow D$ e $D \rightarrow S$)

$M \rightarrow D$ (Perché già $M \rightarrow D$)

$MP \rightarrow M$ (Perché M compare a primo membro)

Risultato dei 3 passaggi uno dietro l'altro:

Impiegato (Matricola, Cognome, Grado, Retribuzione, Dipartimento, Supervisore, Progetto, Anzianità)

{M → RSDG, MS → CD, G → R, D → S, S → D, MPD → AM}

Eliminazione degli attributi estranei



{M → R, M → S, M → D, M → G, MS → C, MS → D, G → R, D → S, S → D, MPD → A, MPD → M}

Eliminazione di: M → R, M → S, M → D, MP → M



{M → R, M → S, M → D, M → G, M → C, M → D, G → R, D → S, S → D, MP → A, MP → M}



{M → D, M → G, M → C, G → R, D → S, S → D, MP → A}

Normalizzazione dei dati

Ridondanza concettuale → non ci sono duplicazioni dello stesso dato, ma sono memorizzate informazioni che possono essere ricavate da altre già contenute nel DB.

Ridondanza logica → esistono duplicazioni sui dati, che possono generare anomalie nelle operazioni sui dati.

Decomposizione di schemi

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

Esempio di decomposizione

L'intuizione è che si devono "estrarre" gli attributi che sono determinati da attributi non chiave ovvero "creare uno schema per ogni funzione"

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

Impiegato → Stipendio

Impiegato	Stipendio
Rossi	20
Verdi	35
Neri	55
Mori	48
Bianchi	48

Progetto → Bilancio

Progetto	Bilancio
Marte	2
Giove	15
Venere	15

Impiegato Progetto → Funzione

Impiegato	Progetto	Funzione
Rossi	Marte	tecnico
Verdi	Giove	progettista
Verdi	Venere	progettista
Neri	Venere	direttore
Neri	Giove	consulente
Neri	Marte	consulente
Mori	Marte	direttore
Mori	Venere	progettista
Bianchi	Venere	progettista
Bianchi	Giove	direttore

Dato $R(T)$ e T_1, \dots, T_k sottoinsiemi di T , $\rho = [R_1(T_1), \dots, R_k(T_k)]$ è una **decomposizione** di R se e solo se $T_1 \cup \dots \cup T_k = T$

Una decomposizione dovrebbe sempre soddisfare le seguenti proprietà:

- La **decomposizione senza perdita**, che garantisce la ricostruzione delle informazioni originarie senza generazione di tuple spurie (che non esistevano precedentemente)
- La **conservazione delle dipendenze**, che garantisce il mantenimento dei vincoli di integrità (di dipendenza funzionale) originari
- **Soddisfacimento della FNBC**: ogni tabella prodotta deve essere in Forma Normale.

La **chiave** è l'unico modo per avere una decomposizione senza perdita di informazione

Uno schema $R(X)$ si decompone senza perdita dei dati negli schemi $R_1(X_1)$ ed $R_2(X_2)$ se, per ogni possibile istanza r di $R(X)$, il join naturale delle proiezioni di r su X_1 ed X_2 produce la tabella di partenza.

Teorema

la decomposizione senza perdita è garantita se l'insieme degli attributi comuni alle due relazioni ($X_1 \cap X_2$) è chiave per almeno una delle due relazioni decomposte

X_1, X_2 sottoinsiemi di X dove $X_1 \cup X_2 = X$

$$X_0 = X_1 \cap X_2$$

Si decompone senza perdita su X_1 e X_2 se:

- $X_0 \rightarrow X_1$
Oppure
- $X_0 \rightarrow X_2$

EX.

$$R = \{A, B, C, D\}$$

$$F = \{A \rightarrow B, B \rightarrow C, A \rightarrow D\}$$

$$R_1 = \{A, B, D\}$$

$$R_2 = \{B, C\}$$

$$R_1 \cup R_2 = \{A, B, C, D\} = R \text{ ok!}$$

$$R_0 = R_1 \cap R_2 = \{A, B, D\} \cap \{B, C\} = \{B\}$$

Controllo se B è chiave per R_1 o per R_2

$$R_1^+ = \{B, A\} \rightarrow \text{no superchiave}$$

$$R_2^+ = \{B, C\} \rightarrow \text{superchiave}$$

Quindi R_0 è chiave di $R_2 \rightarrow$ *Decomposizione senza perdita di informazioni*

La **conservazione delle dipendenze** è cruciale per mantenere la semantica di un database dopo una decomposizione. Un problema comune si verifica quando una dipendenza funzionale, valida nello schema originale, non può essere verificata localmente nelle relazioni decomposte, rendendo necessaria la ricombinazione delle relazioni per controllarla. Questo accade, ad esempio, se una dipendenza coinvolge attributi distribuiti su più relazioni senza che vi sia una relazione che la contenga integralmente.

Proiezione delle Dipendenze

Dato $R < T, F >$, e $T_1 \subseteq T$, la proiezione di F su T_1 è:

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

Dato $F = \{A \rightarrow B, B \rightarrow C, C \rightarrow D\}$ e $T_1 = \{A, B, C\}$, si proietta:

- $A \rightarrow B$: sì (tutti gli attributi in T_1).
- $B \rightarrow C$: sì (tutti gli attributi in T_1).
- $C \rightarrow D$: no ($D \notin T_1$).

Risultato: $\pi_{T_1}(F) = \{A \rightarrow B, B \rightarrow C\}$.

ⓘ Teorema

sia $\rho = \{R_i < T_i, F_i >\}$ una decomposizione di $R < T, F >$ che preserva le dipendenze e tale che T_j per qualche j sia una superchiave per $R < T, F >$. Allora ρ preserva i dati.

Forme Normali

Esistono diversi tipi di forme normali:

- **1FN**: Impone una restrizione sul tipo di una relazione, ogni attributo ha un tipo elementare, no righe o colonne ripetute
- **2FN, 3FN, FNBC**: Impongono restrizioni sulle dipendenze

Una relazione r è in **forma normale di Boyce e Codd (BCNF o FNBC)** se, per ogni dipendenza funzionale (non banale) $X \rightarrow Y$ definita su di essa, X contiene una chiave K di r (è una superchiave)

La relazione sugli Impiegati

[Impiegato](#) | [Stipendio](#) | [Progetto](#) | [Bilancio](#) | [Funzione](#)

non è in forma normale di Boyce and Codd perché esiste la dipendenza funzionale

Impiegato → Stipendio (**Impiegato non è (super)chiave per la relazione**)

Impiegato NON è superchiave perché nell'esempio lo è solo Impiegato, Progetto:

infatti Impiegato da solo non mi permette di identificare univocamente tutte le nuple della relazione

• Ad esempio, in

StudentiEdEsami([Matricola](#), Nome, Provincia, AnnoNascita, [Materia](#), Voto)

• Matricola → Nome e Matricola non è (super)chiave.

• il Nome dipende dalla Matricola che non è chiave.

Anche in questo caso non è FNBC perché solo Matricola

non mi identifica ogni tupla di questa tabella

Teorema

$R < T, F >$ è in BCNF $\Leftrightarrow \forall X \rightarrow A \in F$ non banale, X è superchiave

L'idea è di partire dallo schema contenente tutti gli attributi e di dividerlo fino a quando non otteniamo uno schema che sia in BCNF

$R < A, B, C, D, E >$

$F = \{ CE \rightarrow A, D \rightarrow E, CB \rightarrow E, CE \rightarrow B \}$

1) CONTROLLO $CE \rightarrow A$

$CE^+ = CEA\beta \rightarrow \boxed{\text{NO SK!}}$

$\Rightarrow R_1 = CEA\beta$

$R_2 = R - CE^+ + CE = CED$

2) PROGETTO LE DIPENDENZE SU $R_1 \in R_2$

$R_1 = CEA\beta \quad F_1 = \{ CE \rightarrow A, CB \rightarrow E, CE \rightarrow B \}$

$R_2 = CED \quad F_2 = \{ D \rightarrow E \}$

3) CONTROLLO LE DIPENDENZE SU R_1 E R_2

$$R_1 \rightarrow CE^+ = CEA\beta$$

$$CB^+ = CBEA$$

SK!

SK!

} R_1 OK!

$$R_2 \rightarrow D^+ = DE$$

NO SK!

MANCA C

↳ SCOMPONGO ULTERIORMENTE R_2

$$R_3 = DE$$

$$R_h = R_2 - D^+ + D = CDE - DE + D = CD$$

5) PROGETTO LE DIPENDENZE

$$R_3 = DE$$

$$F_3 = \{D \rightarrow E\}$$

$$R_h = CD$$

$$F_h = \{ \}$$

⚠ Attenzione!

L' algoritmo per la BCNF può non preservare le dipendenze

3FN

Non sempre è possibile ottenere una decomposizione BCNF.

La terza forma normale 3NF è un target di normalizzazione che permette di ottenere automaticamente:

- Decomposizioni senza perdita
- Decomposizioni che preservano tutte le dipendenze

Una relazione r è in 3FN se:

$\forall X \rightarrow Y$ non banale, è verificata almeno una delle seguenti condizioni:

- X contiene una chiave K di r
- Ogni attributo in Y è contenuto in almeno una chiave K di r

Vantaggi:

è sempre ottenibile da qualsiasi schema di partenza

Svantaggi:

è meno restrittiva della BCNF, tollera alcune ridondanze ed anomalie sui dati e certifica meno la qualità dello schema ottenuto

Teorema

$R < T, F >$ è in 3FN se $\forall X \rightarrow A \in F$ non banale, allora X è una superchiave o A è primo

Algoritmo

Diviso in 5 step:

$$Q = (M, C, G, R, D, S, P, A)$$

$$F = \{ M \rightarrow R, S \rightarrow D, G \rightarrow R, D \rightarrow S, S \rightarrow D, M P \rightarrow A \}$$

① CREARE COPERTURA CANONICA f DI F

1.1 CREO DF ATOMICHE:

$$G = \{ M \rightarrow R, M \rightarrow S, M \rightarrow D, M \rightarrow G, M \rightarrow C, M \rightarrow D, G \rightarrow R, D \rightarrow S, S \rightarrow D, M P \rightarrow A, M P \rightarrow M \}$$

POSSO TORNARE
POSSO TORNARE
BANALE

1.2 NUOVO ATTRIBUTO ESTRATTO E DF RIDONDANTI:

$$G = \{ M \rightarrow D, M \rightarrow G, M \rightarrow C, G \rightarrow R, D \rightarrow S, S \rightarrow D, M P \rightarrow A \}$$

2) DECOMPOGO f IN INSIEMI COR SESSO X DI $X \rightarrow Y$

$$G_1 = \{ M \rightarrow D, M \rightarrow G, M \rightarrow C \}$$

$$G_2 = \{ G \rightarrow R \}$$

$$G_3 = \{ D \rightarrow S \}$$

$$G_4 = \{ S \rightarrow D \}$$

$$G_5 = \{ M P \rightarrow A \}$$

(2b) UNISCO LE DIPENDENZE

$$G_1 = \{ M \rightarrow D \text{ } G \text{ } C \}$$

$$G_2 = \{ G \rightarrow R \}$$

$$G_3 = \{ D \rightarrow S \}$$

$$G_4 = \{ S \rightarrow D \}$$

$$G_5 = \{ MP \rightarrow A \}$$

(3) TRASFORMO OGNI GRUPPO IN UNA RELAZIONE CON IL MEMBRO DI SINISTRA COME CHIAVE

$$G_1 = \{ M \rightarrow D \text{ } G \text{ } C \} \rightarrow R_1 (\underline{M} \text{ } D \text{ } G \text{ } C)$$

$$G_2 = \{ G \rightarrow R \} \rightarrow R_2 (\underline{G} \text{ } R)$$

$$G_3 = \{ D \rightarrow S \} \rightarrow R_3 (\underline{D} \text{ } \underline{S})$$

$$G_4 = \{ S \rightarrow D \} \rightarrow R_4 (\underline{S} \underline{D})$$

$$G_5 = \{ MP \rightarrow A \} \rightarrow R_5 (\underline{MP} \text{ } A)$$

4) Esistono schemi contenuti in A5M?

R_1 (MDC)

R_2 (GR)

R_3 (DS)

R_4 (SD)

R_5 (MPA)

R_3 (DS)

IMDIFFERENTE IN

QUESTO
CASO

5) Conosciamo se esiste una R che contiene una chiave, aumentando insieme $R^{(n+1)}$ con la chiave

Trovò la chiave da G midotto:

$$G = \{ M \rightarrow D, M \rightarrow G, M \rightarrow C, G \rightarrow R, D \rightarrow S, S \rightarrow D, M P \rightarrow A \}$$

$R(MCDGRSPA)$

$M_P \rightarrow$ NON COMPARISCE A DESTRA

$M_P^+ = MPA D G C R S$ ✓ MP È CHIAVE

R IN 3FN = $(R_1(MDG), R_2(a, R), R_3(G, S), R_4(M, PA))$

SQL

SQL

I comandi di interrogazione, o **QUERY**, permettono di effettuare una ricerca dei dati presenti nel database che soddisfano particolari condizioni richieste dall'utente.

Sintassi:

```
SELECT ListaAttributi  
FROM ListaTabelle  
WHERE condizioni  
GROUP BY ListaAttributiRaggruppamento  
HAVING CondizioniAggregate  
ORDER BY ListaAttributiOrdinamento
```

EX.

Rubrica

Matr	Cognome	Nome	Email	tel

```
SELECT Cognome, Nome, tel  
FROM Rubrica
```

Rubrica

Matr	Cogn	Nome	Email	tel
...	...	Pippo
...	...	Mario
...	...	Silvia
...	...	Mario
...	...	Marco
...	...	Mario

```
SELECT * # tutti gli attributi
FROM Rubrica
WHERE Nome="Mario"
```

Studenti

Matricola	Nome	Cognome	Indirizzo

Esami

cod.Materia	Nome	Docente	studente

```
SELECT Matricola, studente
FROM Studenti, Esami # esegue una join per controllare
WHERE Studenti.Matricola = Esami.studente
```

La query considera il prodotto cartesiano tra le tabelle in **ListaTabelle** (**JOIN**). Fra queste seleziona solo le righe che soddisfano la **Condizione** (**SELEZIONE**) e infine valuta le espressioni specificate nella target list **ListaAttributi** (**PROIEZIONE**).

Le condizioni che seguono WHERE sono condizioni booleane quali:

- OP di Confronto: `<, >, =, ≠, ≤, ≥`
- Connettori Logici: `AND, OR, NOT`
- Operatori: `BETWEEN, IN, LIKE, IS NULL`

BETWEEN consente di selezionare le righe con gli attributi che rientrano in un certo range:

```
SELECT nome, Dip
FROM Impiegati
WHERE Stip BETWEEN 1000 AND 1500
```

IN consente di selezionare righe che hanno un attributo che assume valori contenuti in una lista:

```
SELECT *
FROM Insegnanti
WHERE corso IN ("BD", "IS", "CN")
```

LIKE è usato per ricerche "wildcard" di una stringa di valori. Le condizioni di ricerca possono contenere letterali, caratteri o numeri:

- `%`: denota zero o più caratteri.
- `_`: denota un carattere.

```
SELECT *
FROM Persone
WHERE nome LIKE "%a" #nome che termina con "a"
```

```
SELECT *
FROM Persone
WHERE nome LIKE "G___G" #nome che inizia per "G" e dopo 3 caratteri ricompare "G"
```

Esiste anche la forma negativa degli operando appena visti che intuitivamente controlla la condizione opposta:

- NOT BETWEEN
- NOT IN
- NOT LIKE

NULL

IS NULL è un operatore che verifica se il contenuto dell'operando è NULL.
NOT NULL esegue la condizione opposta.

Età = NULL → i valori NULL non sono mai uguali fra loro, non posso usarlo come comparatore non avendo un valore unico.

Età IS NULL → effettua correttamente il controllo che vogliamo.

Ordinamento Operatori Aggregati e Raggruppamento

Si possono ordinare i risultati di una SELECT in base a un attributo in modo crescente o decrescente attraverso le keyword ASC/DESC.

```
SELECT lista_attributi  
FROM nome_tabella  
WHERE codizioni  
ORDER BY Attributo [ASC/DESC] #ASC è default
```

Nella target list possiamo avere espressioni che calcolano valori a partire da insiemi di n-uple. Vengono chiamati Operatori Aggregati e sono:

- COUNT : Restituisce il numero di righe/il numero di valori di un particolare attributo.
- MIN : Calcola il minore degli elementi di una colonna.
- MAX : Calcola il maggiore degli elementi di una colonna.
- AVG : Calcola la media dei valori NON NULL di una colonna.
- SUM : Calcola la somma dei valori di una colonna.

Nelle funzioni di gruppo i valori NULL sono ignorati.

Genitori

Genitore	Figlio
Sergio	Franco
Luigi	Olga
Luigi	Filippo
Franco	Andrea
Franco	Aldo

```
SELECT *  
FROM Genitori  
WHERE Genitore = 'Franco'
```

Genitore	Figlio
Franco	Andrea
Franco	Aldo

count

NumFigliDiFranco
2

```
SELECT count(*)  
as NumFigliDiFranco  
FROM Genitori  
WHERE Genitore = 'Franco'
```

Possiamo usare delle specifiche per selezionare i valori di cui siamo interessati:

- `(*)` : *TUTTE* le righe selezionate
- `ALL` : tutti i valori *NON NULL* delle righe
- `DISTINCT` : tutti i valori *NON NULL DISTINTI* delle righe

Di default viene usato `ALL`.

EsamiBD

Studente	BD	LBD
012345	27	NULL
032456	25	23
035221	NULL	NULL
033445	28	30
032441	NULL	30

```
SELECT count([ALL] BD) "ContaBD", count(distinct LBD) "ContDistLBD"  
FROM EsamiBD
```

contaBD	contaLBD
3	2

Non è possibile utilizzare in una stessa SELECT una proiezione su alcuni attributi e operatori aggregati sulla stessa tabella.

```
SELECT nome, max(reddito)  
FROM persone
```

🚫 Error

L'attributo Nome, però, non è aggregato né raggruppato. Il database non può determinare **quale valore di Nome associare al risultato aggregato**. Questo porta a un errore.

```
SELECT min(età), max(reddito)  
FROM persone
```

✓ Success

È corretta perché la SELECT contiene solo funzioni di aggregazione (MIN e MAX), quindi non ci sono attributi non aggregati che richiederebbero una clausola **GROUP BY**. La query restituirà **una singola riga** con due colonne.

GROUP BY

La clausola **GROUP BY** raggruppa le righe di una tabella in base ai valori di uno o più attributi, consentendo di applicare funzioni di aggregazione a ciascun gruppo creato.



```
SELECT Dipart, AVG(stipendio)  
FROM Impiegati  
GROUP BY Dipart
```

Quando si effettua un raggruppamento, questo deve essere effettuato su tutti gli elementi della target list che non sono operatori aggregati.
Nell'esempio sopra raggruppiamo per **Dipart** perché così creiamo gruppi su

cui eseguire `AVG(stipendio)`. Senza i "gruppi" non avrebbe senso essendo `Dipart` un attributo puro.

```
SELECT s.Nome, avg(e.Voto)
FROM Studenti s, Esami e
WHERE s.Matricola = e.Matricola
GROUP BY s.Matricola, s.Nome
```

Gli attributi espressi non aggregati nella target list della select (`s.Nome`) devono essere inclusi tra quelli citati nella GROUP BY (`s.Matricola`, `s.Nome`). Gli attributi aggregati devono essere scelti tra quelli non raggruppati.

HAVING

Si possono applicare condizioni sul valore aggregato per ogni gruppo.

```
SELECT dipart, avg(stipendio)
FROM impiegati
GROUP BY dipart
HAVING avg(stipendio) > 1700
```

 Nb

Se la condizione coinvolge un *attributo*, si usa la clausola `WHERE`.

Se coinvolge un *operatore aggregato* si usa la clausola `HAVING`.

JOIN

Se due tavole del database contengono dei dati in comune, possono essere correlate mediante un'operazione di `JOIN`.

Il JOIN fra due tavole è una terza tabella le cui righe sono tutte e sole quelle ottenute dal prodotto cartesiano delle righe delle due tavole di partenza i cui valori delle colonne espresse dalla condizione di join sono uguali.

In SQL il `JOIN` viene realizzato mediante una particolare forma del `SELECT`, in cui

- Nella clausola `FROM` vengono indicate le due tabelle coinvolte
- Nella clausola `WHERE` viene espresso il collegamento fra le due tabelle, mediante la condizione di `JOIN`

```
tab1(A1,A2), tab2(B1,B2)
SELECT tab1.A1, tab2.B1 #tab1.* e tab2.* prendono tutte le colonne
FROM tab1, tab2
WHERE tab1.A2 = tab2.B2
```

Inner join

È un operazione di `JOIN` dove la condizione non è necessariamente una condizione di uguaglianza.

```
SELECT rep.nome
FROM impiegati AS imp reparti AS rep
WHERE imp.IdReparto ≠ rep.IdReparto AND imp.cognome = 'Rossi' and
imp.nome = 'Mario';
```

Self join

È una join dove viene messa una tabella in relazione con se stessa. Si può ottenere ridenominando 2 volte la stessa tabella con nomi diversi.

```
SELECT X.A1, Y.A3
FROM tab1 X, tab1 Y
WHERE X.A2 = Y.A4
```

Clausola JOIN

```
SELECT Attributi
FROM tab1 JOIN tab2 ON CondizioneDiJoin
```

Se dobbiamo operare una `Equi-Join`, ossia un join la cui condizione sia una condizione di uguaglianza, e che sia anche un `Natural Join`, ossia un join

creato su tutte le colonne che hanno il medesimo nome in entrambe le tabelle, possiamo utilizzare la seguente sintassi

```
SELECT ListaAttributi  
FROM tab1 NATURAL JOIN tab2
```

Using

La clausola USING richiede che entrambe le tabelle abbiano una colonna (o un insieme di colonne) con lo stesso nome.

```
SELECT colonne  
FROM tabella1 JOIN tabella2 USING (colonna_comune);
```

OUTER JOIN

```
SELECT lista_attributi  
FROM Tab1 LEFT [OUTER] JOIN Tab2
```

```
SELECT lista_attributi  
FROM Tab1 RIGHT [OUTER] JOIN Tab2
```

```
SELECT lista_attributi  
FROM Tab1 FULL [OUTER] JOIN Tab2
```

[OUTER] è opzionale.

Può essere seguito da ON + <predicato> e USING + <colonne>.

SUBQUERY

Una Subquery è un comando SELECT, racchiuso tra parentesi tonde, inserito all'interno di un comando SQL, per esempio un'altra SELECT.

Ne esistono di 3 tipi:

- *Subquery Scalare*: restituisce un valore

- *Subquery di Colonna*: restituisce una colonna
- *Subquery di Tabella*: restituisce una tabella

EX.

Vogliamo trovare tutti i veicoli della categoria “Autovettura”

Categorie	Cod_cat	Nome_cat							
Veicoli	Targa	Cod_mod	Categoria	Cilindrata	Cod_comb.	cav.Fisc	Velocita	Posti	Imm

Versione senza subquery

```
SELECT Veicoli.*
FROM Categorie, Veicoli
WHERE Categoria = Cod_cat AND Nome_cat = "Autovettura"
```

Versione con subquery

```
SELECT *
FROM Veicoli
WHERE Categoria = (SELECT cod_cat
                   FROM Categorie
                   WHERE Nome_cat = "Autovettura")
```

Con l'utilizzo delle Subquery è possibile utilizzare contemporaneamente funzioni di gruppo e funzioni su singole righe.

```
SELECT *
FROM Veicoli
WHERE Cilindrata > (SELECT AVG(Cilindrata)
                      FROM Veicoli)
```

```
SELECT *
FROM Veicoli
WHERE Cilindrata = (SELECT MAX(Cilindrata)
                      FROM Veicoli)
```

Ovviamente le Subquery si possono annidare tra loro.

EX.

Selezionare targa e velocità dei veicoli che appartengono a Modelli prodotti nella Fabbrica FIAT

Veicolic

Targa	Cod_mod*	Categoria	Cilindrata	Cod_comb.	cav.Fisc	Velocita	Posti	Imm
-------	----------	-----------	------------	-----------	----------	----------	-------	-----

Modelli

Cod_Mod	Nome_Mod	Cod_Fab*	Cilind_Media
---------	----------	----------	--------------

Fabbrica

Cod_Fab	Nome_Fab
---------	----------



```
SELECT targa, velocità
FROM Veicoli
WHERE Cod_mod IN (SELECT Cod_mod
                   FROM Modelli
                   WHERE Cod_fab = (SELECT Cod_fab
                                     FROM
                                     Fabbrica
                                     WHERE Nome_fab = "FIAT"))
```

Quantificazione

Esistono 2 tipi di quantificazioni:

- *Universale* (\forall)
- *Esistenziale* (\exists)

EX.

- Gli studenti che hanno preso *sempre* (o solo) 30: *universale*
- Gli studenti che hanno preso qualche (*almeno* un) 30: *esistenziale*
- Gli studenti che *non* hanno preso *qualche* 30 (senza nessun 30): *universale*
- Gli studenti che *non* hanno preso *sempre* 30: *esistenziale*

Universale negata = Esistenziale

Esistenziale negata = Universale

- *ANY*: almeno 1 risultato della query soddisfa la condizione

- **ALL** : tutti i risultati soddisfano la condizione
- **EXISTS** : se la subquery restituisce almeno una tupla

EXISTS verifica se esiste almeno una riga che soddisfa una condizione, senza preoccuparsi del contenuto dei dati.

Query Correlate

```
SELECT s.Nome
FROM Studenti s
WHERE EXISTS (
    SELECT *
    FROM Esami e
    WHERE e.Matricola = s.Matricola AND e.Voto > 27
)
```

Clausola **WHERE EXISTS**:

- La condizione **EXISTS** verifica se la subquery interna restituisce almeno una riga. Se la subquery trova almeno un esame con il voto maggiore di 27 per lo studente corrente, la condizione diventa vera, e quindi lo studente verrà incluso nel risultato.

Subquery:

- Questa subquery verifica, per ogni studente, se esiste almeno una riga nella tabella Esami dove la Matricola dello studente corrisponde alla Matricola nella tabella Esami e il Voto dell'esame è maggiore di 27.

```
SELECT *
FROM Studenti S
WHERE EXISTS (SELECT *
               FROM Studenti S2
               WHERE S2.Nome = S.Nome AND S2.Cognome =
S.Cognome AND
               S2.Matricola ◁ S.Matricola)
```

Quando EXISTS trova **true**, significa che la subquery ha restituito almeno una riga che soddisfa le condizioni. Quindi:

- **La query principale restituirà la riga corrente.** In altre parole, la riga della query principale sarà inclusa nel risultato.

Quando EXISTS trova `false`, significa che la subquery non ha restituito alcuna riga che soddisfi la condizione, quindi:

- **La query principale non restituirà la riga corrente.**

Le forme `=ANY` (equivalentemente `IN`) e `<ALL` (equivalentemente `NOT IN`), forniscono un modo alternativo per realizzare intersezione e differenza dell'algebra relazionale.

`ANY` equivale a `EXISTS`.

```
SELECT s.Nome
FROM Studenti s
WHERE EXISTS (SELECT *
               FROM Esami e
               WHERE e.Matricola = s.Matricola AND e.Voto
> 27)
```

Da `Exists` può diventare →

```
SELECT s.Nome
FROM Studenti s
WHERE s.Matricola = ANY (SELECT e.Matricola
                           FROM Esami e
                           WHERE e.Voto >27)
```

Oppure →

```
SELECT s.Nome
FROM Studenti s
WHERE 27 < ANY (SELECT e.Voto
                  FROM Esami e
                  WHERE e.Matricola = s.Matricola)
```

EX. di ottimizzazione

```
SELECT s.Nome
FROM Studenti s
```

```
WHERE NOT EXISTS (SELECT *
                   FROM Esami e
                   WHERE e.Matricola = s.Matricola
                 AND e.Voto > 21)
```

può diventare →

```
SELECT s.Nome
      FROM Studenti s
     WHERE s.Matricola NOT IN (SELECT e.Matricola
                                FROM Esami e
                                WHERE e.Voto > 21)
```

Unione, Intersezione e Differenza

- **UNION** : Utilizza come operandi le due tabelle risultanti da comandi SELECT e restituisce una terza tabella che contiene tutte le righe della prima e della seconda tabella.

```
SELECT padre
      FROM paternita
UNION [ALL]
SELECT madre
      FROM maternita
```

- **INTERSECT** : utilizza come operandi due tabelle risultanti dai comandi SELECT e restituisce una tabella che contiene le righe comuni alle due tabelle iniziali.

```
SELECT ...
INTERSECT [ALL]
SELECT ...
```

- **EXCEPT** : utilizza come operandi due tabelle ottenute mediante due select e ha come risultato una nuova tabella che contiene tutte le righe della prima che non si trovano nella seconda.

```
SELECT ...
EXCEPT [ALL]
```

SELECT ...

DML

INSERT: Permette di inserire nuovi dati nella tabella.

```
INSERT INTO nome_tabella [(ListaAttributi)]
VALUES (ListaDiValori) | SQLSelect
```

Esami

Corso	Insegnante	Matricola	Voto
-------	------------	-----------	------

```
INSERT INTO Esami (Corso, Matricola, Voto)
VALUES ('DB1', '123456', 27)
```

Corso	Insegnante	Matricola	Voto
DB1	NULL	123456	27

Ai valori non attribuiti nella **ListaDiValori** viene assegnato NULL, a meno che non sia specificato un diverso valore di default. L'inserimento fallisce se NULL non è permesso per gli attributi mancanti (vincolo di colonna).

Non specificare gli attributi equivale a specificare tutte le colonne della tabella.
NB: Si deve rispettare l'ordine degli attributi.

È possibile effettuare un insert prendendo i dati da un'altra tabella. Questo è possibile mediante il comando di interrogazione del database **SELECT**

DELETE: Per eliminare un elemento bisogna individuare quale mediante la clausola WHERE, dove viene stabilita una condizione che individua l'elemento (o gli elementi) da cancellare.

```
DELETE FROM nome_tabella
[WHERE Condizione]
```

OPERAZIONE NON REVERSIBILE

Se la condizione è omessa questa istruzione cancella l'intero contenuto della tabella

La condizione del DELETE può essere effettuata con una SELECT per eliminare più dati contemporaneamente.

UPDATE : Permette di aggiornare i dati all'interno di una tabella.

```
UPDATE Tabella  
SET Attributo = Espr  
WHERE Condizione
```

EX.

```
UPDATE Tabella_Aule  
SET Aula = 7  
WHERE Aula = 3
```