Basi di Dati 1

1-Modello relazionale

Il **dominio** è un insieme, anche infinito, di valori utilizzabili.
Presi
$D_1, D_2, ..., D_k$ domini, non necessariamente diversi, il **prodotto cartesiano**

$$D_1 \times D_2 \times ... \times D_k$$

é l'insieme:

$$\{(v_1, v_2, ..., v_k) | v_1 \in D_1, v_2 \in D_2, ..., v_k \in D_k\}$$

Esempio:

$D_1 = \{a, b\}$

$D_2 = \{x, y, z\}$

$D_1 \times D_2 = \{(a, x), (a, y), (a, z), (b, x), (b, y), (b, z)\}$

Una **relazione** r è un qualsiasi sottoinsieme di un prodotto cartesiano

$$r \subseteq D_1 \times D_2 \times ... \times D_k$$

Questa relazione sarà di grado k.

Data una relazione r, ogni elemento $t \in r$ è una **tupla.**

La tupla contiene numero di elementi pari al grado della relazione.

L'elemento i-esimo di una tupla viene indicato come $t[i]$ oppure $t.i$ con $i \in [1, k]$

Una relazione ha una **cardinalità** pari al numero di tuple che la compongono.

Si inizia a contare da 1

Esempio:

$D_1 = \{\text{white,black}\}$

$D_2 = \{0, 1, 2\}$

$D_1 \times D_2 = \{(\text{white}, 0), (\text{white}, 1), (\text{white}, 2), (\text{black}, 0), (\text{black}, 1), (\text{black}, 2)\}$

La relazione $r = \{(\text{white}, 0), (\text{black}, 1), (\text{black}, 2)\}$ è una relazione di grado 2 e cardinalità 3

La tupla $t = (\text{white}, 0) \implies t[1] = \text{white e } t[2] = 0$

1.1-Relazioni come tabelle

Le relazioni possono essere rappresentate come tabelle, in cui viene dato un nome ad ogni colonna.

Una **attributo** A è una coppia $(A, dom(A))$ in cui A è il nome dell'attributo e $dom(A)$ il suo dominio. Due attributi possono avere stesso dominio ma non stesso nome.

Uno **schema relazionale** è l'insieme di tutti gli attributi che definiscono la relazione associata allo schema stesso:

$$R(A_1, A_2, ..., A_k)$$

Lo schema che descrive la struttura di una relazione è invariante nel tempo.

Dato uno schema R, una tupla in R è una funzione che associa ad ogni attributo A un elemento del $dom(A)$. Con $t[A]$ indichiamo il valore della tupla corrispondente all'attributo A.

Un insieme di tuple r è un'**istanza di una relazione.**

Esempio:

$R = \{(\text{Nome, String}), (\text{Cognome, String}), (\text{Media, Real})\}$

Nome	Cognome	Media
Marco	Casu	28
Simone	Lidonnici	27

La tabella è un'istanza dello schema R.

Se t_2 è la seconda tupla della tabella allora $t_2[\text{Cognome}] = \text{Lidonnici}$

Un insieme di relazioni distinte è uno **schema di un database**:

$$(R_1, R_2, ..., R_n)$$

Un database relazionale è uno schema di un database in cui sono definite le istanze $(r_1, r_2, ..., r_n)$ in cui $\forall i \in [1, n] \implies r_i$ è un'istanza di R_i.

Se abbiamo un insieme X che contiene gli attributi di uno schema $R(X)$ e $Y \subseteq X$ allora $t[Y]$ è l'insieme degli attributi di Y nella tupla t. Questo sottoinsieme viene chiamato **restrizione** di t

1.2-Scrivere tabelle corrette

I collegamenti tra i vari dati in schemi diversi vengono fatti attraverso i valori.

Esempio:

Matricola	Nome	Cognome
2067818	Marco	Casu
2061343	Simone	Lidonnici

Matricola	Voto	Esame
2067818	30	Sistemi
2061343	28	Algoritmi

In alcuni casi i potrebbero mancare delle informazioni, in questi casi si inserisce `NULL` come valore nella tupla, nel campo dell'attributo mancante. `NULL` è un valore che non appartiene a nessun dominio ma può rimpiazzare qualunque valore.

Esempio:

Matricola	Nome	Cognome
NULL	Nadia	Ge
2061343	Simone	NULL

1.2.1-Vincoli di integrità

I dati devono avere dei vincoli de rispettare, questi vincoli possono essere:

- Vincolo di dominio: valori che devono essere per forza contenuti in un insieme specifico
- Vincolo intra-relazionale: sono vincoli tra valori di una stessa tupla
- Vincolo inter-relazionale: sono vincoli tra valori di diverse istanze
- Vincolo di chiave primaria: il valore deve essere diverso da `NULL` e unico negli attributi appartenenti alla chiave
- Vincolo di esistenza: il valore deve per forza essere diversa da `NULL`

Esempio:

Studenti:

Matr	Nome	Cognome
2067818	Marco	Casu
2061343	Simone	Bianco

Voti:

Matr	Esame	Voto	Lode
2067818	Calcolo	30	Si
2061343	Sistemi	27	NULL

Vincolo di dominio: $18 \leq \text{Voto} \leq 30$

Vincolo intra-relazione: $\neg(\text{Voto} = 30) \implies \neg(\text{Lode} = \text{Si})$

Vincolo inter-relazione: La matricola nello schema Voti deve esistere nello schema Studenti.

Vincolo di chiave primaria: La matricola nello schema Studenti deve per forza esistere ed essere unica.

Vincolo di esistenza: Il nome nello schema Studenti deve per forza esistere.

1.2.2-Chiavi primarie

Una chiave di relazione è un attributo o un gruppo di attributi che devono esistere ed appartenere ad una sola tupla.

Un insieme X di attributi dello schema R è una chiave di R se:

1. Per ogni istanza di R non esistono due tuple con gli stessi valori in tutti gli attributi di X, cioè $\forall t_1, t_2 \in r$ se $t_1[X] = t_2[X] \implies t_1 = t_2$.
2. Per ogni sottoinsieme $X' \subset X$ possono esserci delle tuple che hanno tutti gli attributi di X' uguali ma sono diverse, cioè $\forall X' \subset X \; \exists t_1, t_2 | t_1[X'] = t_2[X']$ ma $t_1 \neq t_2$

Esempio:

Lezione	Edificio	Aula	Orario
Basi di Dati	Chimica	1	16 : 00
Algebra	Informatica	2	17 : 00
Probabilità	Matematica	1	16 : 30

$X = \{\text{Edificio, Aula, Orario}\}$ è una chiave primaria perchè due lezioni non possono essere nello stesso edificio, aula e ora. Togliendo però qualsiasi dei tre si possono avere delle lezioni diverse che abbiano gli altri attributi uguali (lezioni nello stesso edificio e aula ma ore diverse, nella stessa aula e ora ma edificio diverso o nello stesso edificio e ora ma aule diverse).

1.2.3-Dipendenze funzionali

Una dipendenza funzionale è un insieme di attributi:

$$X, Y \subseteq R | X \to Y$$

Si dice "X determina Y" in cui X è il determinante e Y è il dipendente.

Esempio:

$\text{Flight(Code,Day,Pilot,Time)}$

Questo schema ha dei vincoli dati dal buonsenso:

- un volo con un codice parte sempre alla stessa
- un volo parte con un solo pilota, in un determinato orario e tempo

Questi vincoli determinano delle dipendenze funzionali:

- $\text{Code} \rightarrow \text{Time}$
- $\{\text{Day}, \text{Pilot}, \text{Time}\} \rightarrow \text{Code}$
- $\{\text{Code}, \text{Day}\} \rightarrow \text{Pilot}$

> Un'istanza r di uno schema R soddisfa la dipendenza funzionale $X \rightarrow Y$ se:
>
> 1. La dipendenza è applicabile su R, cioè $X \subset R \wedge Y \subset R$.
> 2. Due tuple in R che hanno gli stessi attributi X avranno anche gli stessi attributi Y, cioè $\forall t_1, t_2 \in r$ se $t_1[X] = t_2[X] \implies t_1[Y] = t_2[Y]$

2-Algebra relazionale

L'algebra relazionale è una notazione per specifiche domande(dette query) sul contenuto delle relazioni, le operazioni in algebra relazionale hanno una controparte in SQL. Per processare una query il DBMS trasforma SQL in una notazione simile all'algebra relazionale.

Il linguaggio per interrogare un database è un insieme di operatori binari o unitari applicati a una o più istanze per generare una nuova istanza.

2.1-Basi dell'algebra relazionale

L'algebra relazionale essendo un'algebra è composta da un dominio e delle operazioni che danno come risultato un elemento interno al dominio.

In questo caso i domini sono le relazioni, rappresentate come un insieme di tuple e i risultati delle operazioni sono un altro insieme di tuple.

Le query sono domande sull'istanza delle relazioni.

Ci sono 3 tipi di operazioni:

- modificare una relazione: proiezione, selezione e rinomina
- operazioni su insiemi: unione, intersezione e differenza
- combinare tuple di due relazioni: prodotto cartesiano, join e theta-join

2.2-Modifica di una relazione

2.2.1-Proiezione

La proiezione esegue un taglio verticale su una relazione e sceglie un sottoinsieme degli attributi, creando una tabella con solo questi attributi.

$$\underset{A_1,\ldots,A_k}{\pi}(R)$$

Prende solo le colonne di un'istanza con attributi $A_1, ..., A_k$.

Esempio:

Customer:

Nome	C#	Città
Rossi	1	Roma
Rossi	2	Milano
Bianchi	3	Roma
Verdi	4	Roma
Rossi	5	Roma

Eseguendo $\underset{\text{Nome}}{\pi}(\text{Customer})$:

Nome
Rossi
Bianchi
Verdi

Eseguendo $\underset{\text{Nome, Città}}{\pi}(\text{Customer})$:

Nome	Città
Rossi	Roma
Rossi	Milano
Bianchi	Roma
Verdi	Roma

2.2.2-Selezione

La selezione fa un taglio orizzontale all'istanza di una relazione e sceglie tutte le tuple che rispettano una determinata condizione.

$$\underset{C}{\sigma}(R)$$

La condizione C è un'espressione booleana nella forma:

$$A \; \theta \; B \text{ o } A \; \theta \; a$$

In cui:

- θ è un operatore di comparazione, cioè $\{<, >, =, \leq, \geq, \text{ecc...}\}$
- A e B hanno lo stesso dominio
- a è un elemento del dominio di A usato come costante

Esempio:

Customer:

Nome	C#	Città
Rossi	1	Roma
Rossi	2	Milano
Bianchi	3	Roma
Verdi	4	Roma
Rossi	5	Roma

Eseguendo $\underset{\text{Town}=\text{Roma}}{\sigma}(\text{Customer})$:

Nome	C#	Città
Rossi	1	Roma
Bianchi	3	Roma
Verdi	4	Roma
Rossi	5	Roma

2.2.3-Rinomina

La rinomina cambia il nome di un attributo in un'altro.

$$\underset{A_1 \leftarrow A_2}{\rho}(R)$$

L'attributo A_1 viene rinominato in A_2.

2.3-Operazioni insiemistiche

2.3.1-Istanze compatibili

Le operazioni insiemistiche possono essere fatte solo su istanze compatibili.

Due istanze sono compatibili se:

- hanno lo stesso numero di attributi
- gli attributi corrispondenti hanno lo stesso dominio

2.3.2-Unione

L'unione crea una nuova istanza contenente tutte le tuple appartenenti a una delle due istanze unite.

$$r_1 \cup r_2$$

Bisogna stare attenti a non unire istanze con attributi che hanno lo stesso dominio ma che non c'entrano niente(ed esempio età e matricola).

Nel caso i cui gli attributi abbiano nomi diversi vengono presi i nomi della prima relazione.

Esempio:

Teachers:

Nome	Codice	Dipartimento
Rossi	1	Matematica
Bianchi	2	Fisica
Verdi	3	Inglese

Admin:

Nome	Codice	Dipartimento
Esposito	1	Matematica
Perelli	2	Informatica
Verdi	3	Inglese

Eseguendo $\text{AllStaff} = \text{Teachers} \cup \text{Admin}$:

AllStaff:

Nome	Codice	Dipartimento
Rossi	1	Matematica
Bianchi	2	Fisica
Verdi	3	Inglese
Esposito	1	Matematica
Perelli	2	Informatica

In questo caso abbiamo cancellato una riga che contiene (Verdi, 3, Inglese), per sistemarlo potremmo cambiare il codice degli insegnanti in T1,T2, ecc… e quello degli admin in A1,A2, ecc…

2.3.3-Differenza

La differenza crea una nuova istanza contenente tutte le tuple della prima che non sono presenti nella seconda. Si identifica con $-$.

$$r_1 - r_2$$

La differenza al contrario di unione e intersezione non è commutativa.

Slide 35

Esempio:

Students:

Nome	Codice	Dipartimento
Rossi	1	Matematica
Bianchi	2	Inglese
Verdi	3	Informatica

Admin:

Nome	Codice	Dipartimento
Esposito	4	Informatica
Bianchi	2	Inglese
Perelli	5	Matematica

Eseguendo $\text{Students} - \text{Admins}$:

Nome	Codice	Dipartimento
Rossi	1	Matematica
Verdi	3	Informatica

Eseguendo $\text{Admin} - \text{Students}$:

Nome	Codice	Dipartimento
Esposito	4	Informatica
Perelli	5	Matematica

2.3.4-Intersezione

L'intersezione crea una nuova istanza con le tuple in comune alle due istanze.

$$r_1 \cap r_2$$

Può anche essere definita come $r_1 - (r_1 - r_2)$.

2.4-Operazioni di combinazione

2.4.1-Prodotto cartesiano

Il prodotto cartesiano crea una relazione con tutte le possibili combinazioni delle tuple della prima e della seconda relazione.

$$R_1 \times R_2$$

Esempio:

Customer:

Nome	C#	Città
Rossi	1	Roma
Rossi	2	Milano
Bianchi	3	Roma

Order:

O#	CC#	A#	n° pezzi
1	1	2	100
2	2	2	250
3	3	3	50

Eseguendo $\text{Orders} = \text{Customer} \times \text{Order}$:

Nome	C#	Città	O#	CC#	A#	n° pezzi
Rossi	1	Roma	1	1	2	100
Rossi	1	Roma	2	2	2	250
Rossi	1	Roma	3	3	3	50
Rossi	2	Milano	1	1	2	100
Rossi	2	Milano	2	2	2	250
Rossi	2	Milano	3	3	3	100
Bianchi	3	Roma	1	1	2	100
Bianchi	3	Roma	2	2	2	250
Bianchi	3	Roma	3	3	3	50

In questo caso ho collegato dei Customer ad alcuni Order con C# diversi sbagliando e avendo alcun tuple in eccesso, quindi eseguo una selezione solo delle tuple che hanno C# uguale a CC# e poi successivamente una proiezione per eliminare l'attributo CC# ora superfluo.

Eseguendo $\text{Final} = \underset{\neg CC\#}{\pi}(\underset{\text{C\#=CC\#}}{\sigma}(\text{Customer} \times \text{Order}))$:

Nome	C#	Città	O#	A#	n° pezzi
Rossi	1	Roma	1	2	100
Rossi	2	Milano	2	2	250
Bianchi	3	Roma	3	3	50

2.4.2-Concatenazione(join)

Seleziona le tuple nel risultato di un prodotto cartesiano che hanno gli stessi valori negli attributi in comune.

$$r_1 \bowtie r_2 = \underset{XY}{\pi}(\underset{C}{\sigma}(r_1 \times r_2))$$

Dove:

- $C = \{(R_1.A_1 = R_2.A_1) \wedge (R_1.A_2 = R_2.A_2) \wedge ... \wedge (R_1.A_k = R_2.A_k)$
- X è l'insieme degli attributi di r_1
- Y è l'insieme degli attributi di r_2 non in r_1
- Gli attributi doppi vengono automaticamente tolti

Casi speciali:

- Se non ci sono istanze che soddisfano le condizioni il risultato sarà l'insieme vuoto(che è comunque una relazione)
- Se le relazioni non hanno attributi in comune il risultato è semplicemente il prodotto cartesiano

Dobbiamo inoltre stare attenti che attributi con lo stesso nome abbiano lo stesso significato.

Esempio:

Customer:

Nome	C#	Città
Rossi	1	Roma
Bianchi	2	Milano
Verdi	3	Roma

Order:

O#	C#	A#	n° pezzi
1	1	1	100
2	2	2	150
3	3	2	25
1	1	3	200

Eseguendo $\text{Customer} \bowtie \text{Order}$:

Nome	C#	Città	O#	A#	n° pezzi
Rossi	1	Roma	1	1	100
Rossi	1	Roma	1	3	200
Bianchi	2	Milano	2	2	150
Vardi	3	Roma	3	2	25

2.4.3-Theta join

Il θ-join è una variazione del join in cui la condizione di selezione non è per forza il fatto che attributi con nome uguale abbiano valore uguale ma possiamo scegliere noi quali attributi collegare tra loro.

$$r_1 \underset{A\theta B}{\bowtie} r_2 = \underset{XY}{\pi}\left(\underset{A\theta B}{\sigma}(r_1 \times r_2)\right)$$

In questo caso il join non collegherà due attributi con lo stesso nome ma collegherà l'attributo A e B in base alla condizione.

Esempio:

Customer:

Nome	C#	Città
Rossi	1	Roma
Rossi	2	Milano
Bianchi	3	Roma

Order:

O#	CC#	A#	n° pezzi
1	1	2	100
2	2	2	250
3	3	3	50

Eseguendo $\text{Customer} \underset{\text{C\#=CC\#}}{\bowtie} \text{Order}$:

Nome	C#	Città	O#	CC#	A#	n° pezzi
Rossi	1	Roma	1	1	2	100
Rossi	2	Milano	2	2	2	250
Bianchi	3	Roma	3	3	3	50

2.5-Quantificazione universale

Se vogliamo scrivere una query per una quantificazione universale usiamo il simbolo $\forall$ che è equivalente a $\nexists$ l'opposto. Le operazioni hanno sempre quantificazione universale.

$$\forall x \rightarrow \varphi(x) = \nexists x \rightarrow \neg\varphi(x)$$

Bisogna ricordare che il contrario di "sempre" è "esiste almeno un caso in cui è falso" e non "è falso in ogni caso".

$$\neg(\forall x \rightarrow \varphi(x)) \neq \nexists x \rightarrow \varphi(x)$$
$$\neg(\forall x \rightarrow \varphi(x)) = \exists x \rightarrow \neg\varphi(x)$$

Esempi:

Customer:

Nome	C#	Città
Rossi	1	Roma
Rossi	2	Milano
Bianchi	3	Roma

Order:

C#	A#	n° pezzi
1	1	50
1	1	200
2	2	150
3	3	125

Se vogliamo trovare i nomi e città delle persone che hanno **sempre** fatto ordini da più di 100 pezzi dobbiamo eliminare dal totale le persone che hanno fatto anche solo una volta un ordine da meno di 100 pezzi.

$$\underset{\text{Nome,Città}}{\pi}((\text{Customer} \bowtie \text{Order}) - \underset{\text{n° pezzi} \leq 100}{\sigma}(\text{Customer} \bowtie \text{Order}))$$

Employees:

Nome	C#	Sezione	Salario	Supervisor#
Rossi	1	B	100	3
Pirlo	2	A	200	3
Bianchi	3	A	500	NULL
Neri	4	B	150	1

Per trovare i dipendenti con stipendio maggiore dei supervisori bisogna fare prima un join della tabella con una copia di se stessa collegando il codice dei supervisori con i codici dei dipendenti.

Per non confonderci rinominiamo tutti i valori della copia con una C davanti.

$$\text{CEmployees} = \text{Employees}$$

$$\text{Employees2} = \text{Employees} \underset{\text{Employees.Supervisor\#=CEmployees.C\#}}{\bowtie} \text{CEmployees}:$$

Nome	C#	Sez	Sal	Sup#	CNome	CC#	CSez	CSal	CSup#
Rossi	1	B	100	3	Bianchi	3	A	500	NULL
Pirlo	2	A	200	3	Bianchi	3	A	500	NULL
Neri	4	B	150	1	Rossi	1	B	100	3

Successivamente bisogna selezionare solo le tuple in cui lo stipendio dei supervisori è minore dei dipendenti e poi proiettare solo nome, codice e salario del dipendente.

$\text{Final} = \underset{\text{Nome,C\#,Salario}}{\pi}(\underset{\text{Sal}>\text{CSal}}{\sigma}(\text{Employees2}))$

Nome	C#	Salario
Neri	4	150

3-Progettare un database relazionale

Per creare un database dobbiamo analizzare gli attributi che deve contenere il database e capire se è meglio fare uno schema singolo e multipli schemi collegati.

È preferibile dividere gli attributi in più schemi per evitare problemi di inserimento(se alcuni dati possono essere aggiunti successivamente), cancellazione(se bisogna cancellare una tupla ma lasciare l'esistenza di alcuni attributi) o ridondanza.

Per progettare un buon database bisogna separare i concetti in schemi diversi.

Possiamo identificare i concetti rappresentati in una relazione dalla chiave.

3.1-Notazioni

- Uno schema relazionale viene definito con R ed è un insieme di attributi:

  $R = \{A_1, A_2, ..., A_n\}$ oppure $R = A_1, A_2, ..., A_n$
- Un attributo viene definito con le prime lettere dell'alfabeto: $A, B, ...$
- Un insieme di attributi viene definito con le ultime lettere dell'alfabeto: $X, Y, ...$
- L'unione di due insiemi $X \cup Y$ viene definita con XY
- Una tupla t su R è una funzione che associa ad ogni attributo di $A_i \in R$ un valore $t[A_i]$ nel dominio di A_i

3.2-Dipendenze funzionali

Una dipendenza funzionale è un insieme di attributi:

$$X, Y \subseteq R | X \to Y$$

Si dice “X determina Y” in cui X è il determinante e Y è il dipendente.

L’insieme delle dipendenze funzionali di uno schema si identifica con F.

Un’istanza r è legale rispetto ad F se:

$$\forall (X \to Y) \in F \implies r \text{ SAT } (X \to Y)$$

Se $F = \{A \to B, B \to C\}$ allora è come se in F ci fosse anche $A \to C$.

Esempio:

$AB \to C$

A	B	C
1	1	1
1	0	1
1	1	1

Soddisfa la dipendenza perché:

$t_1[AB] = t_3[AB] \implies t_1[C] = t_3[C]$

A	B	C
1	1	1
1	0	1
1	1	0

Non soddisfa la dipendenza perché:

$t_1[AB] = t_3[AB]$ ma $t_1[C] \neq t_3[C]$

3.3-Chiusura di un insieme di dipendenze funzionali

Dato uno schema relazionale R e un insieme di dipendenze funzionali F, la chiusura di F è l’insieme di tutte le dipendenze funzionali soddisfatte da tutte le tuple in R.

Viene denotata con F^+.

$$F^+ = \{X \to Y | \forall r \text{ legale } \implies r \text{ SAT } (X \to Y)\}$$

Da questo possiamo dire che:

$$F \subseteq F^+$$
$$Y \subseteq X \implies (X \to Y) \in F^+$$

3.3.1-Assiomi di Armstrong

Definiamo un altro insieme di dipendenze funzionali F^A creato con degli assiomi partendo da F:

1. $(X \to Y) \in F \implies (X \to Y) \in F^A$ (inclusione iniziale)
2. $Y \subseteq X \subseteq R \implies (X \to Y) \in F^A$ (riflessività)
3. $(X \to Y) \in F^A \implies (XZ \to YZ) \in F^A \;\; \forall Z \in R$ (aumento)
4. $\left.\begin{array}{r}(X \to Y) \in F^A \\ (Y \to Z) \in F^A\end{array}\right\} \implies X \to Z \in F^A$ (transitività)

Da questi primi 4 assiomi si ottengono implicitamente altre 3 proprietà:

1. $\left.\begin{array}{r}(X \to Y) \in F^A \\ (X \to Z) \in F^A\end{array}\right\} \implies (X \to YZ) \in F^A$ (unione)
2. $\left.\begin{array}{r}(X \to Y) \in F^A \\ Z \subseteq Y\end{array}\right\} \implies (X \to Z) \in F^A$ (decomposizione)
3. $\left.\begin{array}{r}(X \to Y) \in F^A \\ (WY \to Z) \in F^A\end{array}\right\} \implies (WX \to Z) \in F^A$ (pseudotransitività)

Dimostrazione unione:

$$\left.\begin{array}{r}(X \to Y) \in F^A \xRightarrow{\text{aumento}} (XX \to XY) \in F^A \\ (X \to Z) \in F^A \xRightarrow{\text{aumento}} (XY \to ZY) \in F^A\end{array}\right\} \xRightarrow{\text{trans}} (X \to YZ) \in F^A$$

Dimostrazione decomposizione:

$$\left.\begin{array}{r}Z \subseteq Y \xRightarrow{\text{rifl}} (Y \to Z) \in F^A \\ (X \to Y) \in F^A\end{array}\right\} \xRightarrow{\text{trans}} (X \to Z) \in F^A$$

Dimostrazione pseudo-transitività:

$$\left.\begin{array}{r}(X \to Y) \in F^A \xRightarrow{\text{aumento}} (WX \to WY) \in F^A \\ (WY \to Z) \in F^A\end{array}\right\} \xRightarrow{\text{trans}} (WX \to Z) \in F^A$$

3.4-Chiusura di un insieme di attributi

Dato uno schema relazionale R con F insieme delle dipendenze su R e $X \subset R$ la chiusura di X rispetto ad F definita come X_F^+ o X^+:

$$X_F^+ = \{A|(X \to A) \in F^A\}$$

Ovviamente:

$$X \subseteq X_F^+ \qquad \text{(riflessività)}$$

Dato uno schema relazionale R con F insieme delle dipendenze funzionali su R:

$$(X \to Y) \in F^A \iff Y \subseteq X^+$$

Dimostrazione:

$Y = A_1, A_2, ..., A_n$

- $Y \subseteq X^+ \implies (X \to A_i) \in F^A \quad \forall A_i \in Y \xRightarrow{\text{unione}} (X \to Y) \in F^A$
- $(X \to Y) \in F^A \xRightarrow{\text{decomp}} (X \to A_i) \in F^A \quad \forall A_i \in Y \implies A_i \in X^+ \; \forall i \implies Y \subseteq X^+$

3.4.1-Calcolare X⁺

Dato uno schema relazionale R con F insieme della dipendenze funzionali, X^+ si calcola tramite un algoritmo:

```
def CalcolaChiusura(schema: R, dipendenze: F, attributi: X):
    Z = X
    S = {A | ∃(Y → V)∈ F, Y ⊆ Z, A ∈ V }
    while S ⊄ Z:
        Z = Z ∪ S
        S = { A | ∃(Y → V)∈ F, Y ⊆ Z, A ∈ V }
    return Z
```

Esempio:

$R = ABCDEH$

$F = \{AB \to CD, EH \to D, D \to H\}$

Seguendo l'algoritmo calcoliamo la chiusura di A, D, AB:

```
Z = A
S = A
while S ⊄ Z:  #A è sottoinsieme di A
return Z = A
```

```
Z = D
S = H
while S ⊄ Z:  #H non è sottoinsieme di D
    Z = DH
    S = H
while S ⊄ Z:  #H sottoinsieme di DH
return Z = DH
```

```
Z = AB
S = CD
while S ⊄ Z:  #CD non è sottoinsieme di AB
    Z = ABCD
    S = CDH
while S ⊄ Z:  #CDH non è sottoinsieme di ABCD
    Z = ABCDH
    S = CDH
while S ⊄ Z:  #CDH è sottoinsieme di ABCDH
return Z = ABCDH
```

3.4.2-Z=X$^+$

Dato un insieme di attributi X, l'insieme Z uscito dall'algoritmo è uguale alla chiusura di X:

$$Z = X^+$$

Dimostrazione tramite doppia conclusione:

Definisco:

$$Z_i = \begin{cases} Z \text{ dopo } i \text{ cicli while} \implies Z_{i+1} = Z_i \cup S_i \\ Z_f = Z_{\text{finale}} \end{cases}$$

1. $Z_f \subseteq X^+$

   Dimostro per induzione che se $Z_i \subseteq X^+$ allora anche $Z_{i+1} \subseteq X^+$:

   - Caso base: $i = 0$

     $$Z_0 = X \subseteq X^+$$

   - Passo induttivo:

     $$Z_i \subseteq X^+ \implies Z_{i+1} \subseteq X^+$$

   - $\forall A \in Z_{i+1}$ ci sono due casi possibili:

     1. $A \in Z_i \xRightarrow{\text{passo induttivo}} A \in X^+$
     2. $A \in S_i \implies \begin{cases} \exists (Y \to V) \in F \\ Y \subseteq Z_i \\ A \in V \end{cases} \implies Y \subseteq Z_i \subseteq X^+ \implies (X \to Y) \in F^A \xRightarrow{\text{trans}} (X \to V) \in F^A \implies V \subseteq X^+ \implies A \in X^+$

2. $X^+ \subseteq Z_f$

   $$R = \begin{array}{c|c|c|} & Z_f & R \backslash Z_f \\ \hline t_1 & 11...1 & 01...0 \\ \hline t_2 & 11...1 & 10...1 \\ \hline \end{array}$$

   (con t_1 e t_2 uguali su Z_f)

   Devo controllare se l'istanza è legale per ogni dipendenza di F:

   $\forall (V \to W) \in F$ ci sono due casi possibili:

- $V \cap R \backslash Z_f \neq 0 \implies t_1[V] \neq t_2[V]$
- $V \subseteq Z_f \implies W \subseteq S_f \subseteq Z_f \implies t_1[W] = t_2[W]$

Ora che ho dimostrato che l'istanza è legale:

$$\forall A \in X^+ \implies (X \to A) \in F^A \implies X = Z_0 \subseteq Z_f \implies A \in Z_f$$

3.5-F^+=F^A

Dato uno schema relazionale R con F insieme delle dipendenze funzionali:

$$F^+ = F^A$$

Dimostrazione tramite doppia conclusione:

1. $F^A \subseteq F^+$

   Presa una dipendenza funzionale $(X \to Y) \in F^A$ definisco:

$$F_i^A = \begin{cases} \text{dipendenze funzionali} \\ \text{ottenute con } i \\ \text{applicazioni degli assiomi} \\ \text{di Armstrong} \end{cases}$$
$$F_0^A \subseteq F_1^A \subseteq F_2^A \subseteq ... \subseteq F^A$$

   Per induzione devo dimostrare che se una dipendenza ottenuta con i applicazioni degli assiomi di Armstrong appartiene ad F^+ allora anche la dipendenza ottenuta con $i + 1$ applicazioni degli assiomi di Armstrong appartiene ad F^+:

   - Caso base: $i = 0$

     $F_0^A = F \subseteq F^+$

   - Passo induttivo:

$$F_i^A \subseteq F^+ \implies F_{i+1}^A \subseteq F^+$$

   - L'ultimo passaggio(per passare da i a $i + 1$) può essere qualsiasi dei 3 assiomi:
     - Riflessività:

$$Y \subseteq X \implies (X \to Y) \in F_{i+1}^A$$

$$t_1[X] = t_2[X] \implies t_1[Y] = t_2[Y] \implies (X \to Y) \in F^+$$

     - Aumento:

$$(V \to W) \in F_i^A \implies (VZ \to WZ) \in F_{i+1}^A$$

$$t_1[VZ] = t_2[VZ] \implies \begin{cases} t_1[V] = t_2[V] \\ t_1[Z] = t_2[Z] \end{cases} \implies$$
$$\begin{cases} t_1[W] = t_2[W] \\ t_1[Z] = t_2[Z] \end{cases} \implies t_1[WZ] = t_2[WZ] \implies$$
$$(VZ \to WZ) \in F^+$$

- Transitività:

$$\left.\begin{array}{l}(X \to Y) \in F_i^A \\ (Y \to Z) \in F_i^A\end{array}\right\} \implies (X \to Z) \in F_{i+1}^A$$

$$t_1[X] = t_2[X] \implies t_1[Y] = t_2[Y] \implies t_1[Z] = t_2[Z] \implies (X \to Z) \in F^+$$

2. $F^+ \subseteq F^A$

$$r = \text{istanza legale} = \underbrace{\begin{array}{c|c|c|} & X^+ & R\backslash X^+ \\ \hline t_1 & 11...1 & 01...0 \\ \hline t_2 & 11...1 & 10...1 \\ \hline \end{array}}_{\text{uguali su } X^+}$$

Devo controllare se l'istanza è legale per ogni dipendenza di F:

$\forall(V \to W) \in F$ ci sono due casi possibili:

- $V \cap R\backslash X^+ \neq 0 \implies t_1[V] \neq t_2[V]$
- $\left.\begin{array}{r}V \subseteq X^+ \implies (X \to V) \in F^A \\ (V \to W) \in F^A\end{array}\right\} \xRightarrow{\text{trans}} (X \to W) \in F^A \implies W \subseteq X^+$

Ora che ho dimostrato che l'istanza è legale:

$$\forall(X \to Y) \in F^+ \implies Y \subseteq X^+ \implies (X \to Y) \in F^A$$

3.6-Chiavi

Una chiave è un insieme di attributi $X \subseteq R$ in cui:

- $(X \to R) \in F^+$
- $\nexists X' \subseteq X | (X' \to R) \in F^+$

In SQL se $|X| > 1$ bisogna scegliere un attributo come chiave primaria.

4-Terza forma normale

Dato uno schema relazionale R con F insieme delle dipendenze funzionali su R, R è in terza forma normale(**3NF**) se:

$$\forall (X \rightarrow A) \in F^+, A \notin X$$

Una dipendenza è in 3NF se X contiene una chiave o Y è contenuto in una chiave.

$$X \supseteq K \vee Y \subseteq K$$

Per controllare se uno schema è scritto in 3NF devo controllare tutte le dipendenze ottenute decomponendo quelle in F in modo che il dipendente sia un attributo singolo, basta che una non vada bene e tutto lo schema non è scritto in 3NF.

Possiamo dire anche che uno schema è in 3FN se non contiene dipendenze parziali né dipendenze transitive:

$\begin{cases} F = \{AB \rightarrow C, B \rightarrow C\} \\ K = AB \end{cases} \implies B \rightarrow C$ è una dipendenza parziale perché dipende parzialmente da una chiave

$\begin{cases} F = \{A \rightarrow B, B \rightarrow C\} \\ K = A \end{cases} \implies B \rightarrow C$ è una dipendenza transitiva perché dipende in modo transitivo da una chiave

Esempi:

$R = ABCD$

$F = \{A \rightarrow B, B \rightarrow CD\}$

A è l'unica chiave perché determina B e per transitività anche CD

Devo controllare:

- $A \rightarrow B$: va bene perché A è chiave
- $B \rightarrow C$: non va bene perché B non è chiave e C non è contenuto in una chiave
- $B \rightarrow D$

$R = ABCD$

$F = \{AC \rightarrow B, B \rightarrow AD\}$

AC e BC sono chiavi

Devo controllare:

- $AC \rightarrow B$: va bene perché AC è chiave

- $B \to A$: va bene perché A è contenuto nella chiave AC
- $B \to D$: non va bene perché B non è chiave e D non è contenuto in una chiave

4.1-Forma normale di Boyce-Codd

> Dato uno schema relazionale R con F insieme delle dipendenze funzionali, R è in Boyce-Codd NF se tutti i determinanti contengono una chiave.
>
> $$(X \to Y) \in F^+ | K \subseteq X$$
>
> Tutti gli schemi in Boyce-Codd NF sono anche in 3NF ma non è sempre vero il contrario.
>
> A differenza della 3NF non è sempre possibile decomporre uno schema non in BCNF in sottoschemi in BCNF preservando tutte le dipendenze.

4.2-Trasformare uno schema in 3NF

> Uno schema non in 3NF può essere decomposto in un insieme di schemi in 3NF che devono rispettare delle regole:
>
> - devono preservare le dipendenze funzionali applicate ad ogni istanza dello schema originale(preservare F)
> - bisogna poter ricostruire tramite join naturale ogni istanza legale dello schema originale senza aggiungere informazioni(join senza perdita)

Esempio:

$F = \{A \to B, B \to C\}$ non in forma normale perché $B \to C$ è una dipendenza transitiva

Posso scomporlo in due modi:

$$\begin{cases} R_1 = AB \text{ con } F_1 = \{A \to B\} \\ R_2 = BC \text{ con } F_2 = \{B \to C\} \end{cases}$$

$$\begin{cases} R_1 = AB \text{ con } F_1 = \{A \to B\} \\ R_2 = AC \text{ con } F_2 = \{A \to C\} \end{cases}$$

Il secondo modo non va bene perché quando li mettiamo insieme la dipendenza $B \to C$ non è soddisfatta.

4.3-Preservare F

Dato uno schema relazionale R, una decomposizione è un insieme di sottinsiemi di R tali che:

$$R = \bigcup_{i=1}^{k} R_i$$

Questa non è una partizione di R perché i sottoschemi non sono disgiunti.

Presa un'istanza r l'istanza di un sottoschema r_i:

$$r_i = \underset{R_i}{\pi}(r)$$

Dato uno schema relazionale R, una decomposizione $\rho = R_1, ..., R_k$ preserva F se:

$$F \equiv G = \bigcup_{i=0}^{k} \underset{R_i}{\pi}(F)$$
$$\underset{R_i}{\pi}(F) = \{(X \to Y) \in F^+ | XY \subseteq R_i\}$$

Una singola dipendenza $(X \to Y) \in F$ è preservata se $Y \subseteq X_G^+$.

4.3.1-Teorema sulla chiusura

Dati due insiemi di dipendenze funzionali F, G:

$$F \subseteq G^+ \iff F^+ \subseteq G^+$$

Dimostrazione tramite doppia conclusione:

1. $F^+ \subseteq G^+ \implies F \subseteq G^+$

   $$F \subseteq F^+ \subseteq G^+ \implies F \subseteq G^+$$

2. $F \subseteq G^+ \implies F^+ \subseteq G^+$

   Scrivo
   $g \xrightarrow{A} f$ per dire che posso ottenere f applicando gli assiomi di Arstrong su g

   $$\forall f \in F \implies \exists g | g \xrightarrow{A} f \implies G \xrightarrow{A} F \xrightarrow{A} F^+ \implies F^+ \subseteq G^+$$

4.3.2-Calcolare X^+ di G

Dato uno schema relazionale R con F insieme della dipendenze funzionali e G rispetto ad F
X^+ si calcola tramite un algoritmo(non conosciamo G):

```
def CalcolaChiusura(schema: R, dipendenze: F, attributi: X, d
    Z = X
    S = {}
    for i in range(1,k):
        S = S ∪ (Z ∩ Rᵢ)⁺F ∩ Rᵢ
    while S ⊄ Z:
        Z = Z ∪ S
        for i in range(1,k):
            S = S ∪ (Z ∩ Rᵢ)⁺F ∩ Rᵢ
    return Z
```

Dimostrazione per doppia conclusione:

1. $Z_f \subseteq X_G^+$
    - Caso base: $i = 0$

      $Z_0 = X \subseteq X_G^+$
    - Passo induttivo:

      $$Z_i \subseteq X_G^+ \implies Z_{i+1} \subseteq X_G^+$$
    - $\forall A \in Z_{i+1}$ ci sono due casi possibili:
        1. $A \in Z_i \xRightarrow{\text{passo induttivo}} A \in X_G^+$
        2. $A \in S_i \implies A \in [(Z_i \cap R_j)_F^+ \cap R_j] \implies \begin{cases} A \in R_j \\ A \in (Z_i \cap R_j)_F^+ \end{cases} \implies$

           $(Z_i \cap R_j \to A) \in F^A \implies (Z_i \cap R_j \to A) \in \underset{R_j}{\pi}(F) \subseteq G \implies$

           $\left.\begin{array}{l}(X \to Z_i \cap R_j) \in G^A \\ (Z_i \cap R_j \to A) \in G^A\end{array}\right\} \xRightarrow{\text{trans}}$

           $(X \to A) \in G^A \implies A \in X_G^+$

2. $X_G^+ \subseteq Z_f$

   $X \subseteq Z_f \implies X_G^+ \subseteq (Z_f)_G^+$, se supponiamo di conoscere G possiamo applicare l'algoritmo per calcolare X_F^+ ma su G

```
Z = ZF
S = {A | ∃(Y → V)∈ G, A ∈ V, Y ⊆ Z }
while S ⊈ Z:  #se un elemento è in S è sicuramente anche i
return Z = ZF
```

Ogni elemento $A \in S$ è anche in Z perché:

Se esiste

$(Y \to V) \in G \implies YV \subseteq R_i \implies Y \subseteq (R_i \cap Z_f) \xRightarrow{\text{rifl}}$
$(R_i \cap Z_f \to Y) \in G^A \xRightarrow{\text{decomp}} (R_i \cap Z_f \to A) \in G^A \implies$
$A \in (R_i \cap Z_f)^+ \cap R_i \implies A \in Z_f$

Quindi $Z_f = (Z_f)^+_G \implies X^+_G \subseteq Z_f$

4.3.3-F⁺=G⁺

Dati due insiemi di dipendenze funzionali F, G:

$$F \equiv G \iff F^+ = G^+$$

G e F non sono uguali.

$$\left.\begin{matrix} F^+ \subseteq G^+ \\ G^+ \subseteq F^+ \end{matrix}\right\} \iff \begin{cases} F \subseteq G^+ \\ G \subseteq F^+ \end{cases}$$

Dimostrazione tramite doppia conclusione:

1. $G \subseteq F^+$

$$G = \bigcup_{i=0}^{k} \pi_{R_i}(F) \implies G \subseteq F^+$$

2. $F \subseteq G^+$

   per dimostrarlo utilizziamo un algoritmo:

```
def CalcolaSottoinsieme(schema: R, dipendenze: F, attribut
    for (X → Y) in F:
        do X⁺ɠ
        if Y ⊄ X⁺ɠ:
            return False
    return True
```

4.4-Trovare le chiavi di uno schema

Per trovare la chiavi bisogna trovare un attributo o un insieme di attributi tali che la sua chiusura sia R e non ci siano sottoinsiemi tali che la loro chiusura sia R:

$$X = K \iff X^+ = R$$
$$\nexists X' \subseteq X | (X')^+ = R$$

Ci sono delle chiusure che si possono saltare:

- Se un attributo non è determinato da nulla appartiene a tutte le chiavi
- Se un attributo non determina niente, ma è solo determinato non appartiene a nessuna chiave

Per provare a calcolarle in modo facile seguiamo dei passaggi:

1. Calcoliamo $X_i = R - (W - V)$ per ogni dipendenza $V \to W$
2. Calcoliamo $X = (X_1 \cap ... \cap X_n)$
3. Calcol
   - R allora X è l'unica chiave
   - diverso da R allora X non è chiave ed esiste più di una chiave

4.4.1-Teorema di unicità della chiave

Dato uno schema R, possiamo controllare se una chiave è unica scrivendo:

$$X = \bigcap_{(V \to W) \in F} R - (W - V)$$

Calcoliamo X^+ e se:

- $X^+ = R \implies X$ unica chiave
- $X^+ \neq R \implies$ esiste più di una chiave e X non è superchiave

Esempio:

$R = ABCDE$

$F = \{AB \to C, AC \to B, D \to E\}$

$X = (ABCDE - (C - AB)) \cap (ABCDE - (B - AC)) \cap (ABCDE - (E - D)) = ABDE \cap ACDE \cap ABCD = AD$

$X^+ = AD^+ = ADE \neq R \implies$ esisto più chiavi

4.5-Join senza perdite

Dato uno schema relazionale R, una decomposizione $\rho = R_1, R_2, ..., R_k$ ha un join senza perdita se per ogni istanza r di R:

$$r = \pi_{R_1}(r) \bowtie \pi_{R_2}(r) \bowtie ... \bowtie \pi_{R_k}(r) = m_\rho(r)$$

La decomposizione di R, $m_\rho(r)$ ha delle proprietà:

1. $r \subseteq m_\rho(r)$
2. $\pi_{R_i}(m_\rho(r)) = \pi_{R_i}(r)$
3. $m_\rho(m_\rho(r)) = m_\rho(r) \qquad \text{(idempotenza)}$

Dimostrazione:

1. $\forall t \in r \implies t \in (t[R_1] \bowtie ... \bowtie t[R_k]) \subseteq (\pi_{R_1}(r) \bowtie ... \bowtie \pi_{R_k}(r)) = m_\rho(r)$
2. $\pi_{R_i}(m_\rho(r)) = \pi_{R_i}(r)$
   - $\pi_{R_i}(r) \subseteq \pi_{R_i}(m_\rho(r)) \implies \text{punto 1}$
   - $\pi_{R_i}(m_\rho(r)) \subseteq \pi_{R_i}(r)$

   $\forall t \in \pi_{R_i}(m_\rho(r)) \implies \exists t' \in m_\rho(r) | t'[R_i] = t \implies \exists t_i \in r | t_i[R_i] = t'[R_i] \implies t = t'[R_i] = t_i[R_i] \in \pi_{R_i}(r)$
3. $m_\rho(m_\rho(r)) = \pi_{R_1}(m_\rho(r)) \bowtie \pi_{R_2}(m_\rho(r)) \bowtie ... \bowtie \pi_{R_k}(m_\rho(r)) = \pi_{R_1}(r) \bowtie \pi_{R_2}(r) \bowtie ... \bowtie \pi_{R_k}(r) = m_\rho(r)$

4.5.1-Controllare un join senza perdite

Per controllare se una decomposizione ha un join senza perdite si usa una serie di passaggi:

1. Si costruisce una tabella con $|R|$ numero di colonne e k righe
2. In ogni casella della tabella con colonna j e riga i scriviamo $\begin{cases} a & A_j \in R_i \\ b_i & A_j \notin R_i \end{cases}$
3. Per ogni dipendenza $(X \to Y) \in F$ se ci sono due tuple $t_1[X] = t_2[X]$ ma $t_1[Y] \neq t_2[Y]$ allora se una delle due è uguale ad a allora cambia anche l'altra in a mentre se nessuno dei due è uguale a a allora cambia uno delle due facendola diventare uguale all'altra.
4. Se in qualunque momento una riga avesse tutte a allora possiamo fermarci e sapere che ha un join senza perdite
5. Alla fine del controllo di tutte le dipendenze se nessuna riga ha tutte a allora ripartiamo a controllare su tutte le dipendenze, fino a quando un'intero ciclo di dipendenze non cambia nulla sulla tabella

I passaggi precedenti sono derivati da un’algoritmo:

```
def JoinLoseless(schema: R, dipendenze: F, decomposizione: ρ)
    do r=Tabella
    change=True
    while change==True and ∄t in r|t=="a...a":
        change=False
        for (X → Y) in F:
            if t1[X]==t2[X] and t1[Y]!=t2[Y]:
                change=True
                for A in Y:
                    if t1[A]==a:
                        t2[A]=a
                    else:
                        t1[A]=t2[A]
    if ∃t in r|t="a...a":
        return True
    else:
        return False
```

Dimostrazione per assurdo:

Supponiamo che ρ abbia un join senza perdita $m_\rho(r) = r$ ma r non contenga nessuna tupla con solamente a.

Visto che non succede mai che $a \to b_i$ allora $\exists t_i \in \pi_{R_i}(r) | t_i = "a...a" \; \forall i$, precisamente nella riga corrispondente al sottoschema R_i.
Quindi
$m_\rho(r)$ contiene una riga con tutte a.

Esempio:

$R = ABCD$

$F = \{AB \to C, D \to BC\}$

$\rho = \{ABC, ABD, AD\}$

$$\begin{array}{c|c|c|c|c} r & A & B & C & D \\ \hline ABC & a & a & a & b_1 \\ \hline ABD & a & a & b_2 & a \\ \hline AD & a & b_3 & b_3 & a \\ \hline \end{array} \xRightarrow{AB \to C} \underbrace{\begin{array}{c|c|c|c|c} r & A & B & C & D \\ \hline ABC & a & a & a & b_1 \\ \hline ABD & a & a & b_2 \xrightarrow{1} a & a \\ \hline AD & a & b_3 & b_3 & a \\ \hline \end{array}}_{\text{riga 2 ha tutte } a}$$

Possiamo già fermarci e sapere che ha un join senza perdite(sopra la freccia è indicato in quale ciclo è stato eseguito il cambiamento).

4.6-Copertura minimale

Dato un insieme di dipendenze funzionali F, una copertura minimale di F è un insieme di dipendenze funzionali G tale che:

1. Tutte le dipendenze in G hanno un attributo singolo come dipendente(a destra)
2. $\forall(X \to Y) \in G \nexists X' \subset X | G \equiv [G - \{(X \to Y)\}] \cup \{(X' \to Y)\}$
3. $\forall(X \to Y) \in G \implies G \not\equiv G \backslash \{(X \to Y)\}$

Preso un insieme F possiamo calcolare la copertura minimale tramite 3 passaggi:

1. Con la regola di decomposizione divido tutti i dipendenti ad attributi singoli
2. Controllo se le dipendenze possono essere ridotte, cioè $\exists X' \subset X | (X' \to Y)$, in caso si possa sostituisco la dipendenze $(X \to Y)$ con $(X' \to Y)$, per farlo devo calcolare la chiusura di tutti i sottoinsiemi di X e controllare se contengono Y
3. Controllo per ogni dipendenza se $(X)^+_{F \backslash \{(X \to Y)\}}$ (X^+ calcolato sull'insieme F escludendo la dipendenza stessa)contiene Y, se si elimino la dipendenza $(X \to Y)$

Esempio:

$$F = \{AD \to BG, AG \to DE, B \to DE, G \to CE\}$$

1-Divido tutti i dipendenti

$$F = \{AD \to B, AD \to G, AG \to D, AG \to E, B \to D, B \to E, G \to C, G \to E\}$$

2-Elimino le dipendenze che possono essere ridotte

$$AD \to B \implies \begin{cases} A^+ = A \\ D^+ = D \end{cases} \implies B \notin (A^+ \cup D^+)$$

$$AD \to G \implies \begin{cases} A^+ = A \\ D^+ = D \end{cases} \implies G \notin (A^+ \cup D^+)$$

$$AG \to D \implies \begin{cases} A^+ = A \\ G^+ = CEG \end{cases} \implies D \notin (A^+ \cup G^+)$$

$$AG \to E \implies \begin{cases} A^+ = A \\ G^+ = CEG \end{cases} \implies E \in (A^+ \cup G^+) \implies \cancel{AG \to E}$$

$$F = \{AD \to B, AD \to G, AG \to D, B \to D, B \to E, G \to C, G \to E\}$$

3-Controllo se le dipendenze non sono necessarie

$AD \to B \implies (AD)^+_{F\setminus\{(AD\to B)\}} = ADGCE \implies B \notin (AD)^+_{F\setminus\{(AD\to B)\}}$

$AD \to G \implies (AD)^+_{F\setminus\{(AD\to G)\}} = ADBE \implies G \notin (AD)^+_{F\setminus\{(AD\to G)\}}$

$AG \to D \implies (AG)^+_{F\setminus\{(AG\to D)\}} = AGCE \implies D \notin (AG)^+_{F\setminus\{(AG\to D)\}}$

$B \to D \implies (B)^+_{F\setminus\{(B\to D)\}} = BE \implies D \notin (B)^+_{F\setminus\{(B\to D)\}}$

$B \to E \implies (B)^+_{F\setminus\{(B\to E)\}} = BD \implies E \notin (B)^+_{F\setminus\{(B\to E)\}}$

$G \to C \implies (G)^+_{F\setminus\{(G\to C)\}} = GE \implies C \notin (G)^+_{F\setminus\{(G\to C)\}}$

$G \to E \implies (G)^+_{F\setminus\{(G\to E)\}} = GC \implies E \notin (G)^+_{F\setminus\{(G\to E)\}}$

$F = \{AD \to B, AD \to G, AG \to D, B \to D, B \to E, G \to C, G \to E\}$

Quindi la chiusura minimale G:

$G = \{AD \to B, AD \to G, AG \to D, B \to D, B \to E, G \to C, G \to E\}$

4.7-Algoritmo di decomposizione

Dato uno schema relazionale R con insieme di dipendenze funzionali F esiste sempre una decomposizione ρ tale che:

1. Ogni sottoschema è in 3NF
2. Preserva F
3. Ha un join senza perdita

Per trovare la decomposizione possiamo usare un'algoritmo composto da 3 passaggi:

1. Aggiungere a ρ un sottoschema S con tutti gli attributi che non appaiono in nessuna dipendenza
2. Tolgo questi attributi a R e controllo se c'è una dipendenza che contiene tutti gli attributi rimasti in $R - S$, se c'è aggiungo tutti questi attributi come unico sottoschema a ρ
3. Se non c'è per ogni dipendenza aggiungo come sottoschema tutti gli attributi che la compongono

Questa decomposizione preserva F e i sottoschemi sono in 3NF, ma non è detto che abbia un join senza perdita.

Per avere un join senza perdita un sottoschema deve contenere una chiave, se non c'è allora dobbiamo aggiungerne una.

I passaggi precedenti sono derivati da un algoritmo:

```
def CalcolaDecomposizione(schema: R, copertura minimale: F)
    S={}
    ρ={}
    for each A|∀(X → Y)∈ F, A ∈ X, A ≠ Y:
        S=S ∪ A
    if S ≠ ∅:
        R=R-S
        ρ=ρ ∪ {S}
    if ∃(X → Y)∈ F|X ∪ Y=R:
        ρ=ρ ∪ {R}
    else:
        for each (X → Y)∈ F:
            ρ=ρ ∪ {X ∪ Y}
    return ρ
```

Dimostrazione:

1. Preserva F

$$G=\bigcup_{i=1}^{k}\pi_{R_i}(F)$$

$$\forall(X\to Y)\in F\implies XY\in\rho\implies(X\to Y)\in G\implies F\equiv G$$

2. I sottoschemi sono in 3NF
Dividiamo i 3 casi possibili:
    a. $S\in\rho\implies$ tutti gli attributi fanno parte di una chiave quindi è in 3NF
    b. $R\in\rho\implies$ questa dipendenza ha forma $(R-A)\to A\implies R-A$ è la chiave, quindi $\forall(Y\to B)\in F\implies$
    $$\begin{cases}B\neq A\implies B\in R-A\implies B\text{ primo}\\ B=A\implies Y=R-A\implies Y\text{ superchiave}\end{cases}$$
    c. $XA\in\rho\implies$ essendo F copertura minimale $\implies$
    $$X\text{ è chiave di }XA\implies\forall(Y\to B)\in F|YB\subseteq XA\implies\begin{cases}B=A\implies Y=X\implies Y\text{ superchiave}\\ B\neq A\implies B\in X\implies B\text{ primo}\end{cases}$$

Le combinazioni possibili sono:

- $a + b$
- $a + c$
- c

5-Organizzazione fisica

L'organizzazione fisica di una database è divisa in più settori:

1. Hardware di archiviazione e progettazione fisica
    - Gerarchia di archiviazione
    - Interno di un hard disk
    - Passaggio dai concetti logici a quelli fisici
2. Organizzazione record
    - Puntatori
    - Liste
3. Organizzazione file

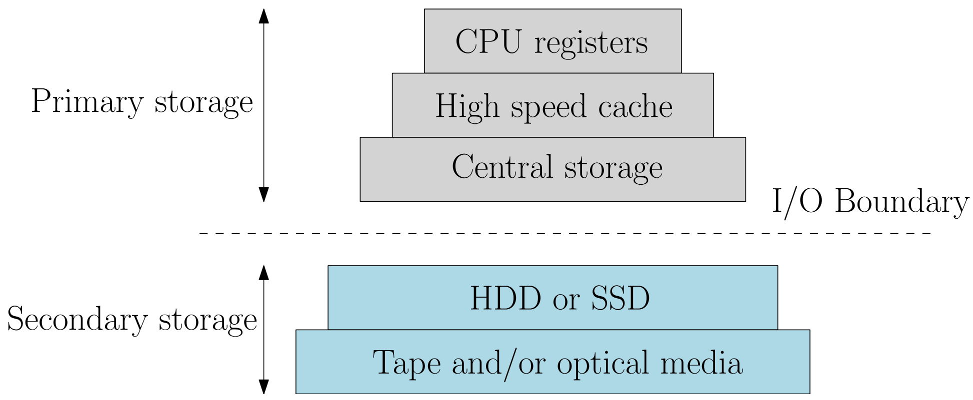
- Organizzazione tramite file heap
- Organizzazione sequenziale
- Organizzazione tramite file hash(casuale)
- Organizzazione sequenziale indicizzata
- Organizzazione in lista
- Indici secondari e file invertiti
- B-trees

5.1-Hardware di archiviazione e progettazione fisica

5.1.1-Gerarchia di archiviazione

L'**Archiviazione primaria** di un DBMS contiene i buffer del database e i codici in esecuzione delle applicazioni e del DBMS. Essa è composta da CPU, Cache e Memoria principale (composta a sua volta da vari banchi di RAM).

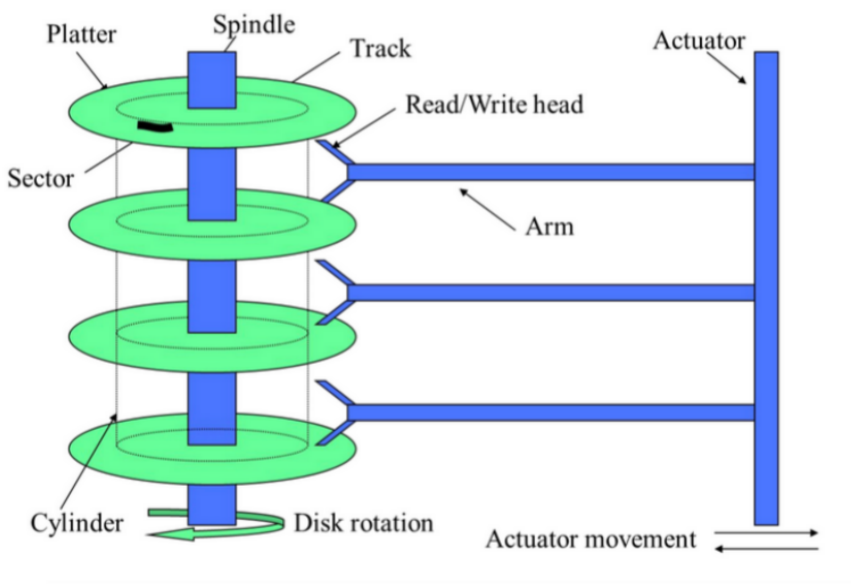
L'**Archiviazione secondaria** si occupa dell'archiviazione permanente dei dati attraverso vari Hard Disk (HDD) e Solid-state drive (SSD), memorizzando i file fisici contenenti i dati del database.



5.1.2-Interno di un hard disk

Un hard disk è costituito da:

- Controller: gestisce gli altri componenti e le richieste di lettura o scrittura, mettendole in coda
- Piatti circolari magnetici collegati ad uno spindle(asse) che gira a velocità costante
- Testine di lettura o scrittura posizionate sui bracci di un attuatore, che si muove per posizione le testine sul punto giusto dei piatti



Per leggere o scrivere un blocco di dati dai piatti bisogna:

- Posizionare l'attuatore, il cui tempo impiegato viene detto **Seek** time
- Attendere la rotazione del disco finché il settore richiesto non si trova sotto la testina di lettura e scrittura, il cui tempo impiegato viene detto **Rotation** time (ROT)
- Trasferire i dati, il cui tempo viene detto **Transfer** time, che dipende dalla grandezza del blocco da leggere(BlS) e dal rateo di trasferimento dei dati(TR), influenzato dalla densità delle particelle magnetiche presenti sul disco e la velocità di rotazione del disco stesso

Il tempo impiegato per completare una lettura o scrittura viene detto **Service** time:

$$\text{Service} = \text{Seek} + \text{ROT} + \frac{\text{BlS}}{\text{TR}}$$

Il tempo impiegato dall'hard disk per restituire la risposta viene detto **Response** time e dipende dal Service time e dal tempo per mettere in coda la richiesta, detto **Queueing** time:

$$\text{Response} = \text{Service} + \text{Queueing}$$

Visto che il tempo do lettura e scrittura dipende dalla posizione delle testine, possiamo dividere il tempo impiegato in due casi:

- **Random Block Access**, cioè il tempo per accedere ad un blocco indipendentemente dalla posizione attuale delle testine

$$\text{T}_{\text{RBA}} = \text{Seek} + \frac{\text{ROT}}{2} + \frac{\text{BlS}}{\text{TR}}$$

- **Sequenzial Block Access**, cioè il tempo per accedere ad un blocco con la testina già posizionata nel settore giusto

$$\text{T}_{\text{SBA}} = \frac{\text{ROT}}{2} + \frac{\text{BlS}}{\text{TR}}$$

5.1.3-Passaggio dai concetti logici a quelli fisici

Dobbiamo trasformare un database dal modello concettuale al modello relazionale e infine alla sua rappresentazione fisica, per capire meglio i termini possiamo usare questa tabella:

Modello concettuale	Modello relazionale	Modello interno
Tipo attributo e valore	Nome colonna e cella	Dati oggetti o campi
Record	Riga o tupla	Record archiviato
Tipo record	Tabella o relazione	File fisico o insieme di dati
Insieme di tipi di record	Insieme di tabelle	Database fisico (collezione di file)
Strutture dati logiche	Chiavi esterne	Strutture di archiviazione

Quindi a livello fisico

- Un database consiste in un insieme di file
- Un file è un insieme di pagine con dimensione fissa

- Una pagine contiene dei record(tuple logiche)
- Un record contiene più campi, contenenti gli attributi delle tuple

5.2-Organizzazione dei file

I file all'interno di un hard disk possono essere organizzati in modo diverso per ottimizzare la velocità con cui si esegue una scrittura o lettura, per far ciò si cerca di aumentare i SBA e di diminuire i RBA più possibile.

5.2.1-Organizzazione tramite file heap

L'organizzazione tramite file heap è il modello più basico, in cui ogni record viene inserito alla fine del file, identificandolo con una chiave di ricerca. L'unico modo per cercare un record è tramite la ricerca lineare, il cui tempo medio avendo NBl blocchi è:

- $\lceil \frac{\mathrm{NBl}}{2} \rceil\ \mathrm{SBA}$ con chiave univoca
- $\mathrm{NBl} \cdot \mathrm{SBA}$ con chiave non univoca

5.2.2-Organizzazione tramite file sequenziali

L'organizzazione tramite file sequenziali inserisce i record in ordine crescente o decrescente in base alla loro chiave di ricerca, in modo che si possa usare anche la ricerca binaria per trovare un record, quindi il tempo di accesso con NBl blocchi è:

- $\lceil \log_2(\mathrm{NBl}) \rceil \mathrm{RBA}$ con ricerca binaria
- $\lceil \frac{\mathrm{NBl}}{2} \rceil \mathrm{SBA}$ con ricerca sequenziale

5.2.3-Organizzazione tramite file hash

L'organizzazione tramite file hash utilizza un algoritmo di hashing(di solito si utilizza la funzione $\bmod$) per collegare la chiave di ricerca all'indirizzo fisico dove si trova il record. Visto che più di un record potrebbe essere mappato allo stesso valore hash, si utilizzano dei bucket per contenere tutti i record mappati nello stesso valore hash. Nella memoria principale viene salvata una bucket directory, dove ogni entrata corrisponde al puntatore al primo blocco del bucket associato.

Nel caso in cui i record mappati ad un bucket superino la capienza del bucket, il record va in overflow e va gestito, solitamente si usa l'indirizzamento aperto, cioè il record viene archiviato al primo posto disponibile.

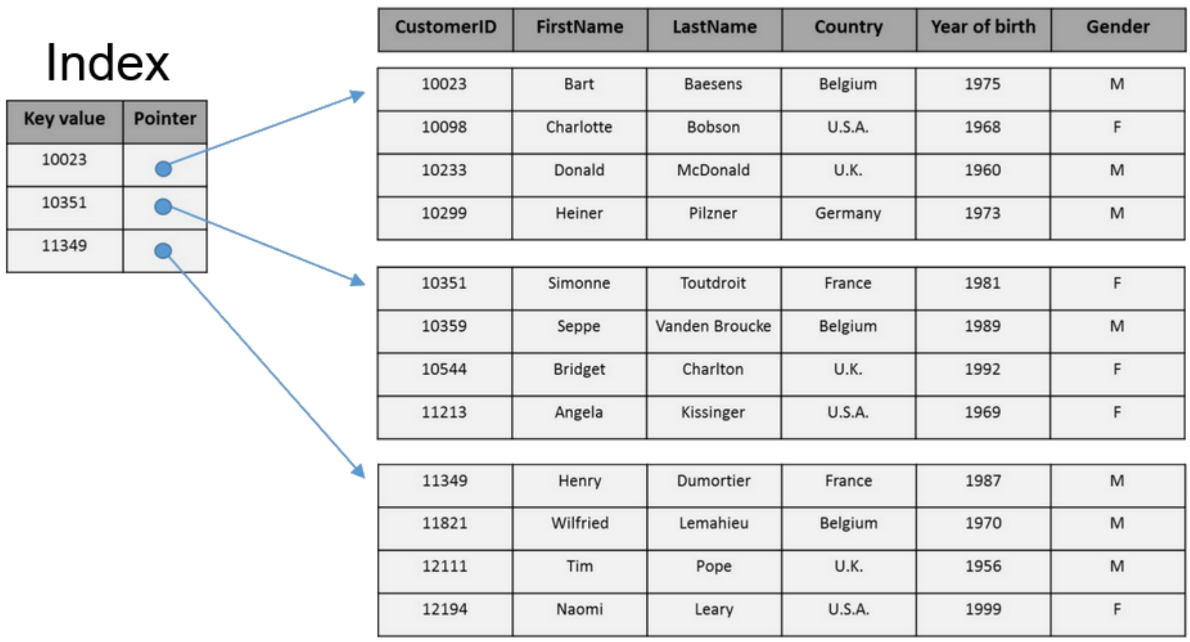
Il tempo di accesso è quindi diverso in base a se il record che ci interessa è in overflow o no:

- Non in overflow: Una RBA per raggiungere il primo blocco del bucket e diverse SBA per trovare il record all'interno del bucket, essendo arciviati in modo sequenziale
- In overflow: Oltre ai passaggi per trovare un record non in overflow dovremo accedere ad altri blocchi in base alla quantità di record in overflow

5.2.4-Organizzazione tramite ISAM

L'organizzazione tramite ISAM ha un file principale diviso in partizioni, ognuna con un'entrata del tipo <Chiave di ricerca del primo record, puntatore indirizzo primo record> archiviata nel file indice. Il file indice può essere di due tipi:

- **Denso**: esiste un'entrata per ogni possibile valore della chiave di ricerca
- **Sparso**: esiste solo l'entrata del primo record di ogni partizione, quindi ogni entrata fa riferimento ad un gruppo di record



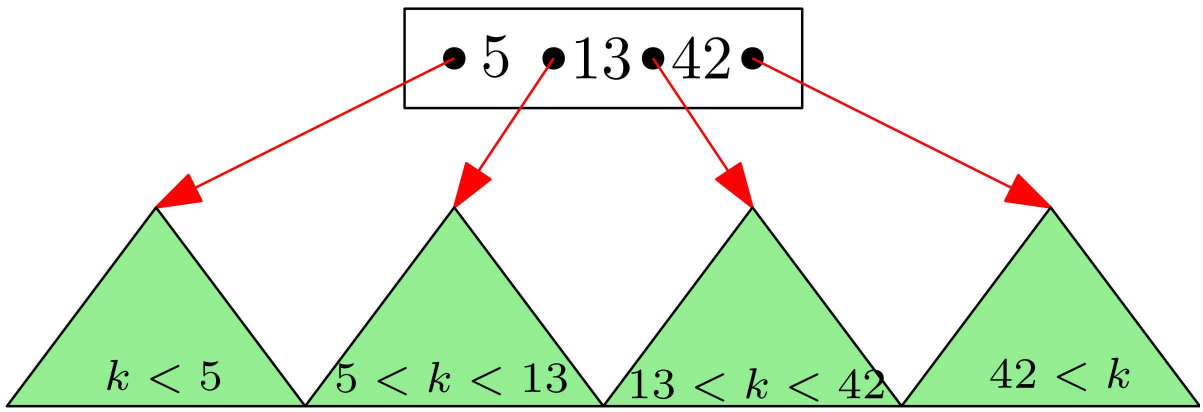
Il tempo di accesso, con NBpFP e NBpFI rispettivamente i blocchi del file principale e del file indice, sarà quindi:

- Ricerca lineare sul file principale: $\text{NBpFP} \cdot \text{SBA}$
- Ricerca binaria sul file principale: $\lceil \log_2(\text{NBpFP}) \rceil \text{RBA}$
- Ricerca binaria tramite file indice: $\lceil \log_2(\text{NBpFI}) \rceil + 1\ \text{RBA}$

5.3.5-Organizzazione tramite B-Tree(bilanciato)

Un **albero di ricerca ad m-vie** è simile ad un albero binario di ricerca, con delle diverse proprietà:

- Un nodo ha da 1 fino a m-1 chiavi
- Il numero di figli va da 0 a k+1, in cui k è il numero di chiavi del nodo, quindi il numero massimo di figli è m
- Le chiavi nel sottoalbero puntato dal puntatore compreso tra due valori k e k' possono avere solo valori compresi tra k e k'



Un **B-Tree** è un albero di ricerca ad m-vie con altre proprietà aggiuntive:

- Ogni nodo ha massimo m figli e m-1 chiavi
- Ogni nodo tranne la radice e le foglie ha minimo $\lceil \frac{m}{2} \rceil$ figli e $\lceil \frac{m}{2} \rceil - 1$ chiavi
- La radice ha minimo due figli tranne se è una foglia
- Tutte le foglie sono allo stesso livello

L'organizzazione tramite B-Tree ha un file indice strutturato ad albero in cui ogni nodo corrisponde ad un blocco, carico sempre per almeno metà e con almeno $\lceil \frac{m}{2} \rceil$ figli.

Ogni blocco ha:

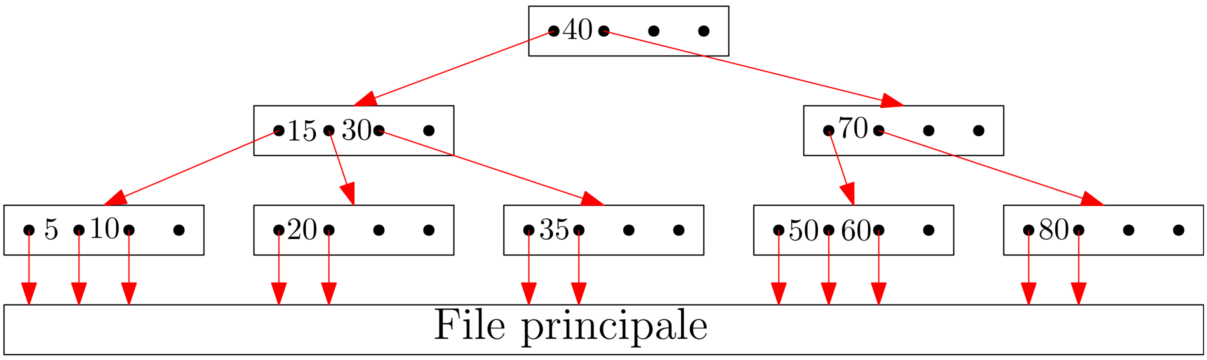
- Delle chiavi di ricerca
- I puntatori ai nodi figli

Nel file principale vengono archiviati i record e tutti i blocchi sono puntati dalle foglie dell'albero. I blocchi del file principale vengono archiviati separatamente e non fanno parte dell'albero.

La ricerca viene effettuata in maniera ricorsiva controllando ogni volta se il valore che stiamo cercando è nel nodo e continuando con il puntatore nell'intervallo corretto. Nel caso in cui il valore che stiamo cercando sia all'interno del nodo sia la chiave K_i il record corrispondente si accede tramite il puntatore ai dati P_{d_i}.

Il tempo di accesso è nel caso peggiore uguale all'altezza dell'albero quindi $O(h)$.

Esempio:



Dato un B-Tree avente n chiavi totali, con m numero massimo di figli e $d = \lceil \frac{m}{2} \rceil$ numero minimo di figli, l'altezza dell'albero sarà:

$$\lceil \log_m(n+1) \rceil \leq h \leq \lfloor \log_d(\frac{n+1}{2}) \rfloor + 1$$

Dimostrazione:

Il numero massimo di chiavi si ottiene se tutti i nodi hanno m figli e $m-1$ chiavi, il numero di chiavi totali allora sarà:

$$n_{max} = (m-1)\sum_{i=0}^{h-1} m^i = (m-1)\frac{m^h - 1}{m-1} = m^h - 1$$

Il numero minimo di chiavi si ottiene se tutti i nodi hanno d figli e $d-1$ chiavi, il numero di chiavi totali allora sarà:

$$n_{min} = 1 + (d-1)(2 \cdot \sum_{i=0}^{h-2} d^i) = 1 + (d-1)(2\frac{d^{h-1} - 1}{d-1}) = 2^{h-1} - 1$$

Da questo possiamo ricavare che:

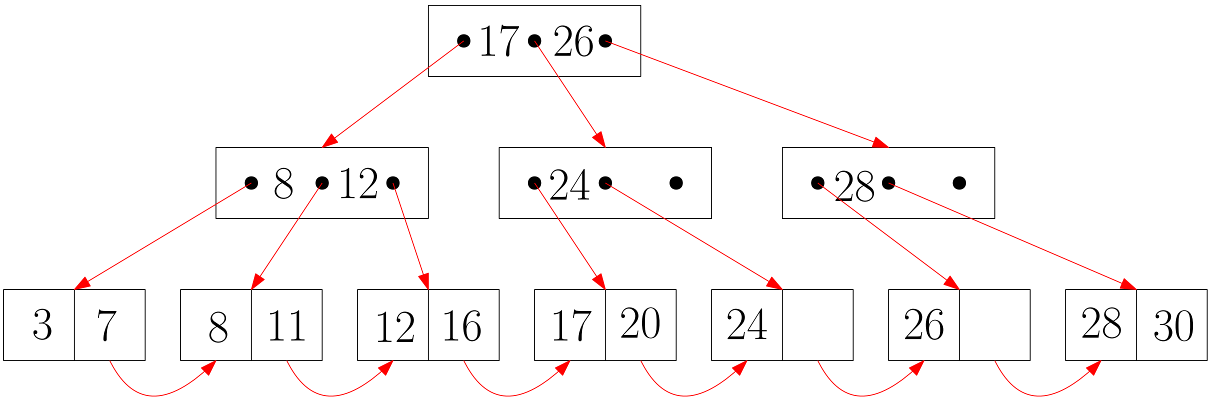
$$2d^{h-1} - 1 \leq n \leq m^h - 1 \implies$$
$$\lceil \log_m(n+1) \rceil \leq h \leq \lfloor \log_d(\frac{n+1}{2}) \rfloor + 1$$

5.3.6-B⁺-Tree

Un B^+-Tree è un B-Tree con delle proprietà aggiuntive:

- Tutti i valori chiave non presenti in un nodo foglia vendono ripetuti in una foglia
- I nodi di livello maggiore hanno sottoinsieme delle chiavi presenti dei nodi foglia
- Ogni foglia ha un puntatore ad una foglia adiacente

Esempio:



6-Concorrenza

In sistemi con una sola CPU i programmi vengono eseguiti in modo interfogliato, cioè che la CPU può:

- Eseguire alcune istruzioni di un programma
- Sospendere quel programma
- Eseguire istruzioni di altri programmi
- Riprendere ad eseguire il programma sospeso

In un DBMS la risorsa a cui i programmi accedono in modo concorrente è il DB. La concorrenza non crea problemi se vengono eseguite solo operazioni di lettura, invece se vengono eseguite anche operazioni di lettura la concorrenza va controllata.

6.1-Transazioni

Una **transazione** è l'esecuzione di una parte di un programma che rappresenta un'unità logica di accesso o modifica del DB.

Le transazioni hanno 4 proprietà, descritte dall'acronimo ACID:

- Atomicità: la transazione non può essere eseguita a metà, deve essere eseguita tutta o non essere eseguita per niente
- Consistenza: la transazione deve iniziare con un DB in stato consistente(non deve violare vincoli) e lo deve lasciare in stato consistente quando è finita l'esecuzione
- Isolamento: la transazione deve essere eseguita in modo indipendente dalle altre e un fallimento non deve interferire con le transazioni successive
- Durabilità: una volta che la transazione ha richiesto un **commit work**, i cambiamenti apportati al DB non devono essere persi, per evitare perdite di dati vengono tenuti dei log che annotano tutte le operazioni sul DB

6.2-Schedule di transazioni

Una schedule di un insieme T di transazioni è l'ordinamento delle operazioni nelle transazioni che conserva l'ordine delle operazioni all'interno delle singole transazioni.

Se una transazione ha un'operazione O_1 precedente ad un'operazione O_2 le due operazioni in una schedule potranno essere separate da operazioni di altre transazioni ma non avremo mai O_2 precedente a O_1.

6.2.1-Schedule seriali

Uno schedule seriale è uno schedule ottenuto permutando le transazioni.

Lo schedule seriale quindi eseguirà le transazioni in maniera sequenziale e non interfogliata. Tutti gli schedule seriali sono corretti.

6.2.2-Serializzabilità e equivalenza di schedule

Uno schedule non seriale è corretto solo se è **serializzabile**, cioè è equivalente(non uguale) ad uno schedule seriale.

Due schedule sono **equivalenti** solo se per ogni dato modificato producono valori uguali, dove due valori sono uguali solo se prodotti dalla stessa sequenza di operazioni.

Esempio:

Schedule 1:

T_1	T_2
$read(X)$	
$X = X + N$	
$write(X)$	
	$read(X)$
	$X = X - M$
	$write(X)$

Schedule 2:

T_1	T_2
	$read(X)$
	$X = X - M$
	$write(X)$
$read(X)$	
$X = X + N$	
$write(X)$	

Le due schedule non sono equivalenti nonostante diano lo stesso risultato X, però sono seriali quindi corrette.

6.2.3-Garantire la serializzabilità

Per garantire la serializzabilità è impossibile determinare in anticipo come verranno eseguite le operazioni e non si possono neanche eseguire e poi cancellare tutte le transazioni se lo schedule non è serializzabile. I sistemi quindi utilizzano metodi che garantiscano la serializzabilità, per evitare il bisogno di testare lo schedule ogni volta.

Per garantire la serializzabilità bisogna:

- Imporre dei **protocolli**, cioè delle regole alle transazioni
- Usare i **timestamp**, degli identificatori delle transazioni generati dal sistema, in base ai quali vengono ordinate le operazioni

6.2.3-Item

Gli **item**, usati dai metodi precedenti, sono unità il cui accesso è controllato e le cui dimensioni sono definite per far si che in media una transazione acceda a pochi item.

La dimensione degli item viene detta **granularità** del sistema.

Una granularità grande permette una gestione efficiente della concorrenza, mentre una granularità piccola permette l’esecuzione concorrente di più transazioni, rischiando però di sovraccaricare il sistema.

6.3-Lock

Un **lock** è un privilegio di accesso ad un item realizzato tramite una variabile associata all'item, che descrive lo stato dell'item rispetto alle operazioni che possono essere eseguite su di esso.

Un lock:

- Viene richiesto da una transazione tramite un'operazione di **locking**, in cui se la variabile di quell'item è **unlocked**(cioè nessuna altra transazione ha un lock su quell'item) la transazione può accedere all'item e assegnare alla variabile il valore **locked**.
- Viene rilasciato tramite un'operazione di **unlocking**, in cui la variabile assegnata all'item viene cambiata da locked a unlocked.

Nel periodo tra l'operazione di locking e quella di unlocking la transazione mantiene un lock sull'item.

6.3.1-Schedule legali

Uno schedule è legale se una transazione effettua un locking ogni volta che legge o scrive un item e rilascia tutti i lock che ha ottenuto.

6.4-Lock binario

Un lock binario ha solo due valori, locked e unlocked.

Le transazioni richiedono l'accesso ad un item X tramite un $lock(X)$ e rilasciano l'item permettendo l'accesso ad altre transazioni tramite $unlock(X)$.

Esempio:

T_1	T_2
$lock(X)$	
$read(X)$	
$X = X + N$	
$write(X)$	
$unlock(X)$	
	$lock(X)$
	$read(X)$
	$X = X + M$
	$write(X)$
	$unlock(X)$
$lock(Y)$	
$read(Y)$	
$Y = Y + N$	
$write(Y)$	
$unlock(Y)$	

6.4.1-Equivalenza di schedule con lock binario

Per verificare l'equivalenza di due schedule con lock binario utilizziamo un modello che non conta le specifiche operazioni ma solo quelle rilevanti per controllare la sequenza, cioè quelle di lock e unlock. Per fare questo associamo ad ogni operazione di $unlock$ una funzione che ha come argomenti gli item letti dalla transazione prima dell'operazione di $unlock$.

Due schedule sono equivalenti se le formule che danno il valore sono uguali per **tutti gli item**.

Esempio:

T_1	Formule
$lock(X)$	
$unlock(X)$	$f_1(X)$
$lock(Y)$	
$unlock(Y)$	$f_2(X,Y)$

T_2	Formule
$lock(Y)$	
$unlock(Y)$	$f_3(Y)$
$lock(X)$	
$unlock(X)$	$f_4(X,Y)$

Schedule non seriale:

T_1	T_2	Formule
$lock(X)$		
$unlock(X)$		$f_1(X_0)$
	$lock(Y)$	
	$unlock(Y)$	$f_3(Y_0)$
$lock(Y)$		
$unlock(Y)$		$f_2(X_0, f_3(Y_0))$
	$lock(X)$	
	$unlock(X)$	$f_4(f_1(X_0), Y_0)$

$X_f = f_4(f_1(X_0), Y_0)$

$Y_f = f_2(X_0, f_3(Y_0))$

Schedule seriali:

T_1, T_2:

T_1	T_2	Formule
$lock(X)$		
$unlock(X)$		$f_1(X_0)$
$lock(Y)$		
$unlock(Y)$		$f_2(X_0, Y_0)$
	$lock(Y)$	
	$unlock(Y)$	$f_3(f_2(X_0, Y_0))$
	$lock(X)$	
	$unlock(X)$	$f_4(f_1(X_0), f_2(X_0, Y_0)))$

$X_f = f_4(f_1(X_0), f_2(X_0, Y_0))$

$Y_f = f_3(f_2(X_0, Y_0))$

T_2, T_1:

T_1	T_2	Formule
	$lock(Y)$	
	$unlock(Y)$	$f_3(Y_0)$
	$lock(X)$	
	$unlock(X)$	$f_4(X_0, Y_0)$
$lock(X)$		
$unlock(X)$		$f_1(f_4(X_0, Y_0))$
$lock(Y)$		
$unlock(Y)$		$f_2(f_4(X_0, Y_0), f_3(Y_0))$

$X_f = f_1(f_4(X_0, Y_0))$

$Y_f = f_2(f_4(X_0, Y_0), f_3(Y_0))$

Quindi lo schedule non seriale non è serializzabile perché il valore finale di X e diverso dal valore finale di tutti gli schedule seriali. Anche quello di Y è diverso ma ne basta uno per dire che non è serializzabile.

6.4.2-Algoritmo per testare la serializzabilità

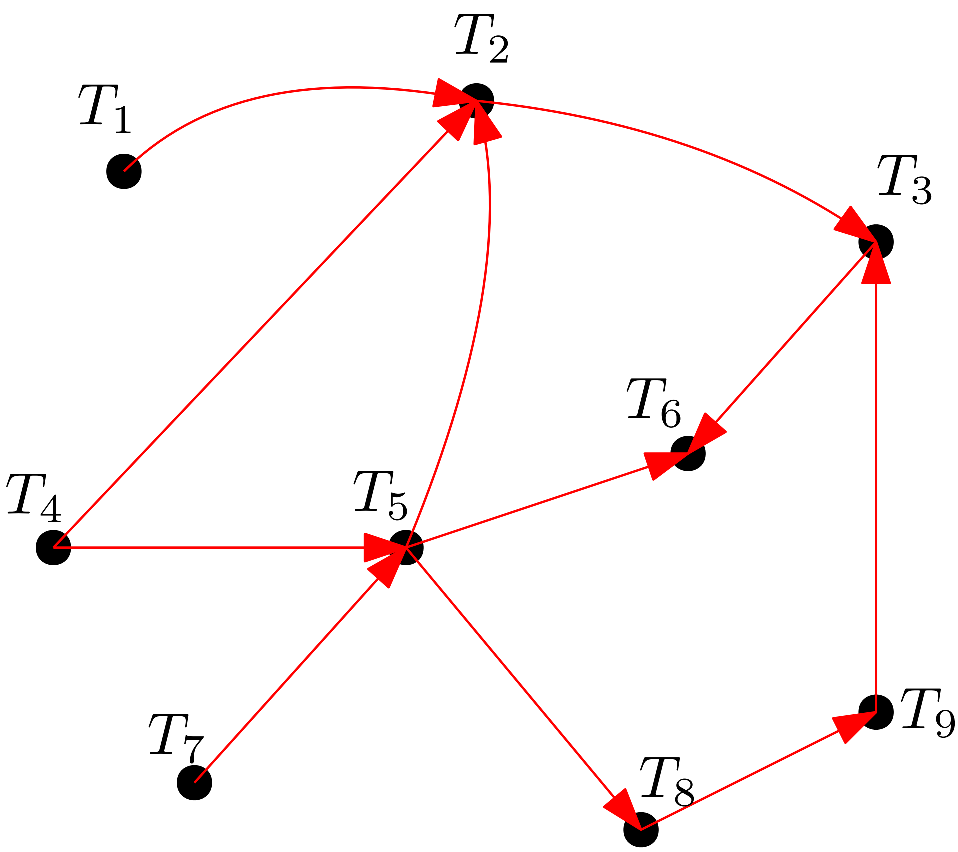
Dato uno schedule S:

1. Crea un grafo G in cui gli archi con l'etichette di un item X tra T_i e T_j indicano che T_j è la prima transazione che fa $lock(X)$ dopo che T_i fa $unlock(X)$
2. Se G ha un ciclo allora S non è serializzabile, sennò basta usare l'ordinamento tipologico per ottenere uno schedule seriale S' equivalente a S. L'ordinamento topologico si ottiene eliminando ricorsivamente i nodi che non hanno archi entranti insieme ai suoi archi uscenti.

Quindi:

$$G \text{ aciclico} \iff S \text{ serializzabile}$$

Esempio:



Il grafo non ha un ciclo quindi è serializzabile e ci sono più ordinamenti topologici possibili tipo:

- $T_1 \rightarrow T_4 \rightarrow T_7 \rightarrow T_5 \rightarrow T_6 \rightarrow T_9 \rightarrow T_2 \rightarrow T_3 \rightarrow T_6$
- $T_4 \rightarrow T_7 \rightarrow T_5 \rightarrow T_1 \rightarrow T_8 \rightarrow T_9 \rightarrow T_2 \rightarrow T_3 \rightarrow T_6$

6.5-Lock a due fasi

Una transazione obbedisce al protocollo di locking a due fasi se prima esegue tutte le operazioni di lock e poi tutte le operazioni di unlock. Il lock a due fasi non è esclusivo del lock a due valori, potremmo avere lock a due fasi con tre valori.

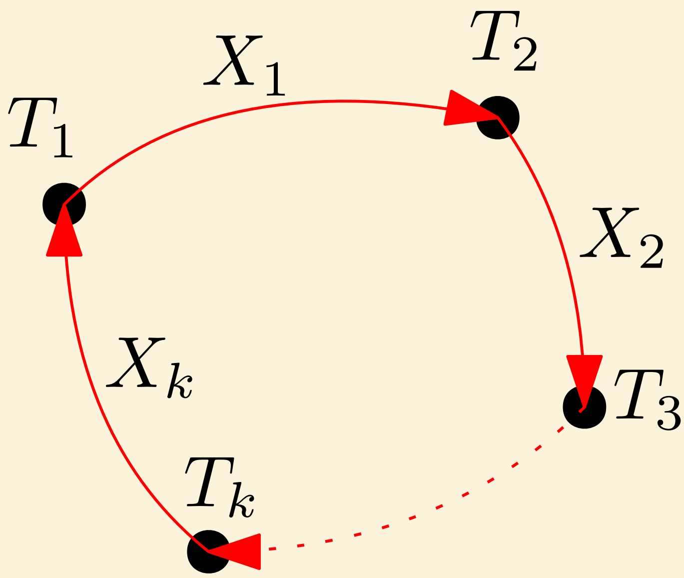
6.5.1-Teorema sul lock a due fasi

Dato un insieme di transazioni T ogni schedule è serializzabile se tutte le transazioni sono a due fasi.

Dimostrazione per assurdo:

Supponiamo di avere uno schedule S composto solo da transazioni a due fasi ma il grado G sia ciclico.

G:



G crea uno schedule S:

T_1	T_2	...	T_k
$unlock(X_1)$			
	$lock(X_1)$		
	$unlock(X_2)$		
		...	
			$unlock(X_k)$
$lock(X_k)$			

T_1 non è a due fasi, contraddizione.

6.6-Lock a tre valori

Un lock a tre valori ha tre valori possibili: rlocked, wlocked e unlocked.

Una transazione che vuole solo leggere un item X esegue $rlock(X)$, impedendo alle altre transazioni di scrivere ma non di leggere. Una transazione che vuole sia leggere che scrivere esegue $wlock(X)$ impedendo alle altre transazioni sia di leggere che di scrivere. Entrambi i lock vengono rilasciati con $unlock(X)$.

Questi lock interagiscono secondo questa tabella:

	X *unlocked*	X *rlocked*	X *wlocked*
T esegue $rlock(X)$	T ottiene un lock di lettura su X variabile $= rlocked$	T ottiene un lock di lettura su X	T aspetta
T esegue $wlock(X)$	T ottiene un lock di scrittura su X variabile $= wlocked$	T aspetta	T aspetta

6.6.1-Equivalenza di schedule con lock a tre valori

Per verificare l'equivalenza di due schedule con lock a tre valori associamo ad ogni operazione di $unlock$ successiva ad un'operazione di $wlock$ una funzione che ha come argomenti gli item letti e scritti dalla transazione prima dell'operazione di $unlock$.

Due schedule sono equivalenti se le formule che danno il valore sono uguali per **tutti gli item su** cui viene effettuato un $wlock$ e vengono letti gli stessi valori ad ogni $rlock$.

6.6.2-Algoritmo per testare la serializzabilità

Dato uno schedule S:

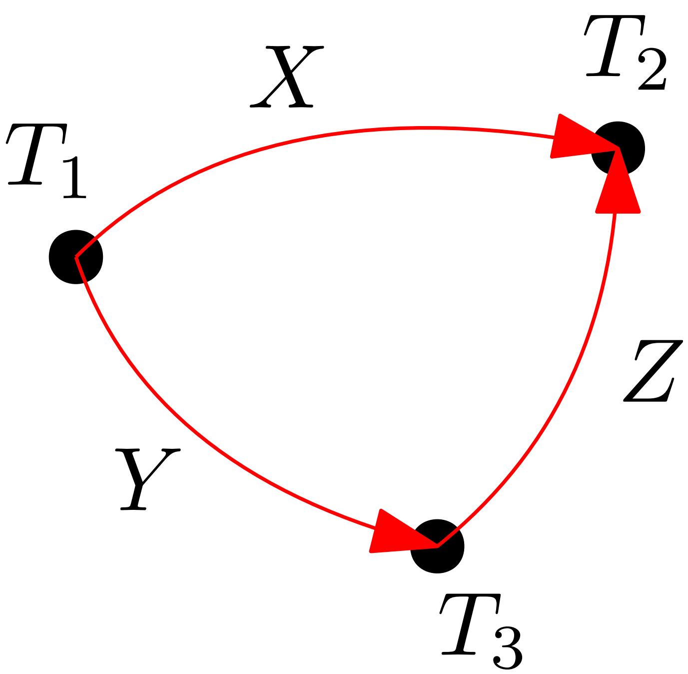
1. Crea un grafo G in cui gli archi con l'etichette di un item X tra T_i e T_j indicano due cose possibili:
   - T_i ha eseguito $rlock(X)$ o $wlock(X)$ e T_j è la prima transazione che fa $wlock(X)$
   - T_i ha eseguito $wlock(X)$ e T_j esegue $rlock(X)$ prima che un'altra transazione esegua $wlock(X)$
2. Se G ha un ciclo allora S non è serializzabile, sennò basta usare l'ordinamento tipologico per ottenere uno schedule seriale S' equivalente a S. L'ordinamento topologico si ottiene eliminando ricorsivamente i nodi che non hanno archi entranti insieme ai suoi archi uscenti.

Esempio:

S:

T_1	T_2	T_3
$rlock(X)$		
$unlock(X)$		
	$wlock(X)$	
	$unlock(X)$	
$wlock(Y)$		
$unlock(Y)$		
		$rlock(Y)$
		$unlock(Y)$
		$wlock(Z)$
		$unlock(Z)$
	$rlock(Z)$	
	$unlock(Z)$	

S crea un grafo G:



Il grafo non ha un ciclo quindi è serializzabile nell'ordinamento topologico:

- $T_1 \rightarrow T_3 \rightarrow T_2$

6.7-Lock write-only e read-only

Un lock write-only e read-only ha tre valori possibili: rlocked, wlocked e unlocked.

Una transazione che vuole solo leggere un item X esegue $rlock(X)$, impedendo alle altre transazioni di scrivere ma non di leggere. Una transazione solo scrivere senza leggere esegue $wlock(X)$ impedendo alle altre transazioni sia di leggere che di scrivere. Entrambi i lock vengono rilasciati con $unlock(X)$.

6.7.1-Equivalenza di schedule con lock write-only e read-only

Per verificare l'equivalenza di due schedule con lock write-only e read-only associamo ad ogni operazione di $unlock$ successiva ad un'operazione di $wlock$ una funzione che ha come argomenti solamente gli item letti dalla transazione prima dell'operazione di $unlock$.

Uno schedule S' è equivalente ad un altro schedule S se:

1. se in S ho T_1 che esegue $wlock(X)$ e dopo T_2 che esegue $rlock(X)$ allora in S' devo mettere prima T_1 e dopo T_2.
2. se T_3 esegue $wlock(X)$ allora in S' posso mettere T_3 prima di T_1 oppure dopo T_2, ma non in mezzo.
3. se T_i esegue $rlock(X)$ e legge il valore inziale di X allora in S' deve precedere qualunque T_j che esegua $wlock(X)$
4. su T_i esegue $wlock(X)$ e scrive l'ultimo valore di X allora in S' deve seguire qualunque T_j che esegua $rlock(X)$

Gli ultimi due vincoli vengono assorbiti dall'1 e dal 2 se aggiungiamo una transazioneT_0 che esegue $wlock()$ su tutti gli item e una transazione T_f che esegue $rlock()$ su tutti gli item.

6.7.2-Algoritmo per testare la serializzabilità

Dato uno schedule S:

1. Crea un poligrafo P in cui gli archi con l'etichette di un item X tra T_i e T_j indicano due cose possibili:
   - T_i ha eseguito $wlock(X)$ e T_j fa $rlock(X)$
   - se T_k esegue $wlock(X)$ su un item X che impone l'esistenza di un arco

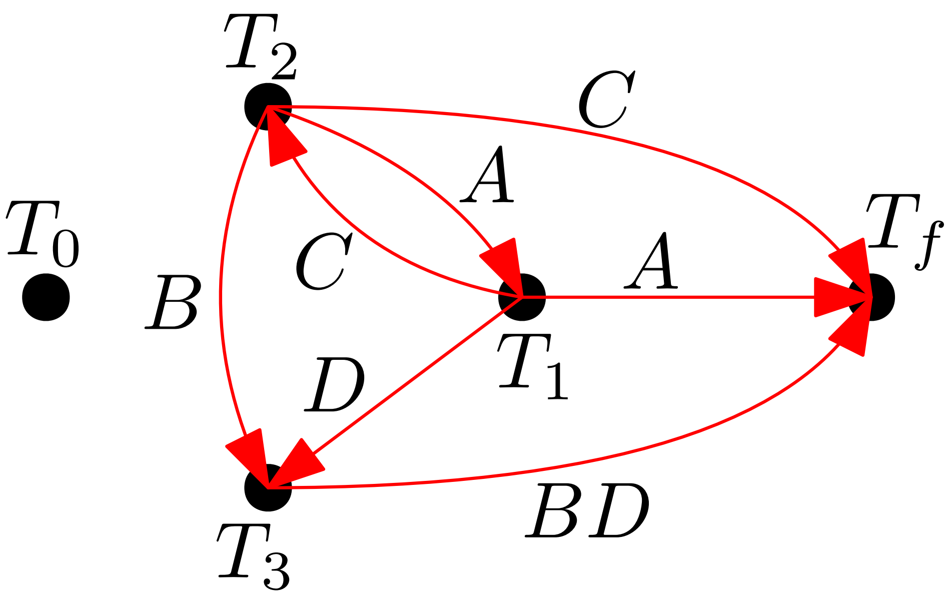
     $T_i \rightarrow T_j$, in questo caso si possono fare 3 cose:
     - creo l'arco $T_j \rightarrow T_k$ se $T_i = T_0 \land T_j \neq T_f$
     - creo l'arco $T_k \rightarrow T_i$ se $T_j = T_f \land T_i \neq T_0$
     - creo due archi alternativi $T_k \rightarrow T_i \land T_j \rightarrow T_k$ se $T_i \neq T_0 \land T_j \neq T_f$
2. Se esiste almeno un grafo P aciclico ottenuto prendendo uno solo tra le coppie di archi alternativi allora S è serializzabile. Per ottenere S' schedule seriale equivalente a S bisogna usare il sorting topologico.

Esempio:

S:

T_1	T_2	T_3
$wlock(A)$		
$unlock(A)$		
	$wlock(C)$	
	$unlock(C)$	
	$rlock(A)$	
	$wlock(B)$	
	$unlock(A)$	
	$unlock(B)$	
$rlock(C)$		
$wlock(D)$		
$unlock(C)$		
$unlock(D)$		
		$wlock(B)$
		$wlock(C)$
		$unlock(B)$
		$unlock(D)$

S crea un grafo G:



Questo grafo ha un ciclo $T_1 \leftrightarrow T_2$ quindi non è serializzabile.

6.8-Deadlock

Un **deadlock** è uno stallo che si verifica quando ogni transazione in un insieme T sta aspettando di ottenere un lock su un item su cui un'altra transazione ha già un lock e quindi rimangono bloccate e potrebbero bloccare anche transazioni non in T.

6.8.1-Verificare un deadlock

Per verificare se sta accadendo una situazione di deadlock si crea un grafo di attesa G:

1. G ha un arco $T_1 \rightarrow T_2$ se T_1 sta aspettando il lock su un item attualmente in lock da T_2
2. Se G ha un ciclo allora c'è una situazione di stallo che coinvolge le transazioni nel ciclo

6.8.2-Soluzioni di un deadlock

Per risolvere un deadlock si utilizza il **roll-back**, cioè presa una transazione nel ciclo:

1. La transazione viene abortita
2. I suoi effetti sul DB vengono annullati
3. Tutti i lock della transazione vengono rilasciati

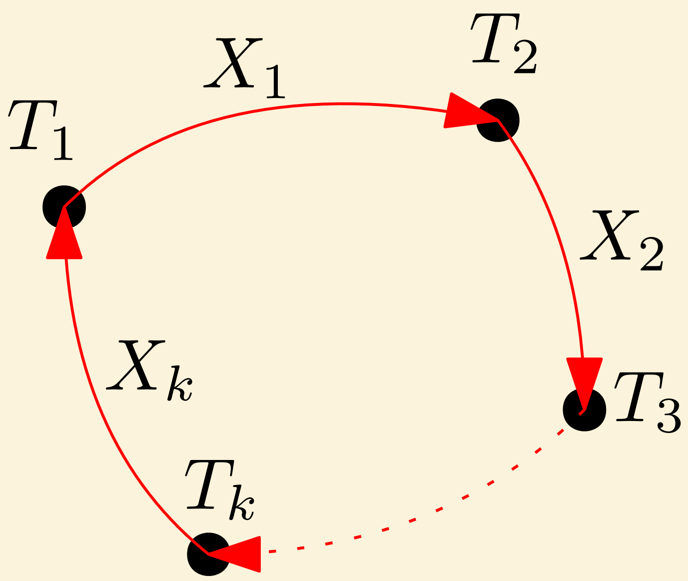
6.8.3-Evitare un deadlock

Per evitare che si verifichi un deadlock si possono utilizzare dei protocolli, come ordinare gli item e imporre alle transazioni di richiedere i lock sugli item in ordine.

In questo modo non ci può essere un ciclo nel grafo di attesa.

Dimostrazione per assurdo:

Supponiamo che e transazioni richiedano gli item in ordine ma nel grafo ci sia un ciclo



Per far apparire il ciclo dovrei avere che X_k precede X_1, ma per l'ordinamento è X_1 a precedere X_k, quindi è una contraddizione.

6.9-Livelock

Un livelock è uno stallo che si verifica quando una transazione aspetta in modo indefinito che gli venga concesso un lock su un item.

6.9.1-Soluzioni di un livelock

SI può evitare che avvenga un livelock in diversi modi come:

- Usare una strategia first came-first served
- Eseguire le transazioni in base alla priorità e aumentare la priorità di una transazione in base al tempo in cui rimane in attesa

6.10-Definizioni varie

6.10.1-Abort di una transazione

Una transazione viene abortita quando si verificano determinate situazioni:

1. La transazione esegue un’operazione non corretta(divisione per 0, accesso non consentito)
2. Lo scheduler rileva un deadlock
3. Lo scheduler fa abortire la transazione per garantire la serializzabilità
4. Si verifica un malfunzionamento hardware o software

6.10.2-Roll-back a cascata

Quando abortiamo una transazione dobbiamo annullare i cambiamenti al DB effettuati sia dalla transazione che a tutte quelle che abbiano letto dati sporchi.

6.10.3-Punto di commit

Il punto di commit di una transazione è il momento in cui la transazione ha ottenuto tutti i lock e ha effettuato tutti i calcoli. Non può quindi essere abortita per i motivi 1 e 3.

6.10.4-Dati sporchi

I dati sporchi sono dei dati scritti da una transazione sul DB prima che abbia raggiunto il punto di commit. Per evitare la lettura di dati sporchi si usa un protocollo più restrittivo rispetto a quello a due fasi.

6.11-Locking a due fasi stretto

Una transazione obbedisce al protocollo di locking a due fasi stretto se:

- La transazione non scrive sul DB fino a quando non ha raggiunto il punto di commit
- Non rilascia lock finchè non ha finito di scrivere sul DB

6.12-Tipi di protocolli

I protocolli si dividono in due tipi:

- Conservativi: cercano di evitare le situazioni di stallo

- Aggressivi: cercano di eseguire le transazioni i più velocemente possibile anche se potrebbe portare a situazioni di stallo

In base alla probabilità che due transazioni richiedano un lock su uno stesso item conviene usare tipi di protocolli diversi:

- Probabilità alta: conviene usare un protocollo conservativo per evitare il sovraccarico dovuto alla gestione dei deadlock
- Probabilità bassa: conviene usare un protocollo aggressivo per evitare il sovraccarico sulla gestione dei lock

6.12.1-Protocolli conservativi

Un protocollo conservativo vuole che una transazione T richieda tutti i lock che servono all'inizio e li ottiene solo se sono tutti disponibili, se anche uno solo non può essere ottenuto la transazione viene messa in attesa. Questo protocollo evita i deadlock ma non i livelock.

Se si volessero evitare anche i livelock, bisogna che la transazione T oltre che ottenere i lock solo se sono tutti disponibili, li ottenga solo se nessuna transazione precedente nella coda sia in attesa di un lock richiesto da T.

I vantaggi dei protocolli conservativi sono:

- Evitare il verificarsi di deadlock e livelock

Gli svantaggi sono:

- L'esecuzione di una transazione può essere ritardata
- La transazione deve chiedere un lock su qualsiasi item potrebbe servirgli anche se poi on lo usa

6.12.2-Protocolli aggressivi

Un protocollo aggressivo permette a una transazione di richiedere un lock prima di leggerlo o scriverlo.

6.13-Timestamp

> 🔖 Un timestamp è un valore che viene assegnato ad una transazione dallo scheduler quando la transazione inizia e può essere il valore di un contatore o l'ora di inizio della transazione. Un timestamp minore implica che la transazione è iniziata prima di un'altra.

6.13.1-Serializzabilità

Uno schedule è serializzabile se è equivalente allo schedule seriale in cui le transazioni sono eseguite in ordine in base al timestamp. Questo vuol dire che per ciascun item acceduto da più transazioni, l'ordine con cui queste lo fanno è in base al timestamp.

Esempio:

T_1:

T_1
$read(X)$
$X = X + 10$
$write(X)$

$TS(T_1) = 110$

T_2:

T_2
$read(X)$
$X = X + 5$
$write(X)$

$TS(T_2) = 100$

Questo schedule sarà quindi serializzabile solo se equivalente allo schedule seriale T_2T_1.

6.13.2-Read timestamp e write timestamp

Viene assegnato ad ogni item X due timestamp:

- Read timestamp $read\_TS(X)$: il più grande tra i timestamp delle transazioni che hanno letto X
- Write timestamp $write\_TS(X)$: il più grande tra i timestamp delle transazioni che hanno scritto X

6.13.3-Algoritmo per controllare la serializzabilità

Ogni volta che una transazione cerca di leggere o scrivere X bisogna controllare che il timestamp della transazione sia maggiore del read timestamp e write timestamp di X.

Se T vuole eseguire $write(X)$:

1. se $read\_TS(X) > TS(T) \implies T$ rolled-back
2. se $write\_TS(S) > TS(T) \implies$ non viene effettuata la scrittura
3. se nessuna della due precedenti si verifica allora $write(X)$ viene eseguita e viene cambiato il write timestamp$X write_TS(X) = TS(T)$

Se T vuole eseguire $read(X)$:

1. se $write\_TS(X) > TS(T) \implies T$ rolled-back
2. se la precedente non si verifica allora $read(X)$ viene eseguita e se $read\_TS(X) < TS(T)$ viene cambiato il read timestamp$X read_TS(X) = TS(T)$

Esempio:

S:

T_1	T_2	T_3
$TS(T_1) = 110$	$TS(T_2) = 100$	$TS(T_3) = 105$
	$read(X)$	
		$read(Y)$
	$X = X + 5$	
$read(X)$		
$X = X + 10$		
		$Y = Y - 5$
$write(X)$		
		$write(Y)$
	$write(X)$	
$read(Y)$		
$Y = Y + 20$		
$write(Y)$		

Il controllo della serializzabilità durante l'esecuzione:

T_1	T_2	T_3	Operazione	Esito
$TS = 110$	$TS = 100$	$TS = 105$		
	$read(X)$		$R\_TS(X) = 100$	
		$read(Y)$	$R\_TS(Y) = 105$	
	$X = X + 5$			
$read(X)$			$R\_TS(X) = 110$	
$X = X + 10$				
		$Y = Y - 5$		
$write(X)$			$W\_TS(X) = 110$	
		$write(Y)$	$W\_TS(Y) = 105$	
	$write(X)$		$W\_TS(X) > TS(T_2)$	Roll-back T_2
$read(Y)$			$R\_TS(Y) = 110$	
$Y = Y + 20$				
$write(Y)$			$W\_TS(Y) = 110$	

E2-Esercizi sulle dipendenze funzionali

E2.1-Trovare le chiavi di uno schema

Dato uno schema relazionale R per trovare le chiavi dobbiamo trovare un attributo o insieme di attributi X tali che $X^+ = R$

Ci sono alcuni attributi che possiamo considerare o escludere direttamente:

- Se un attributo non è determinato da nulla o non appare in nessuna dipendenza allora appartiene a tutte le chiavi
- Se un attributo non determina nulla allora non appartiene a nessuna chiave

Esempio:

$R = ABCDEFG$

$F = \{AC \to DE, C \to FB, BC \to EG, B \to A, A \to CG, B \to C, F \to EG\}$

D, E, G non determinano nulla quindi non apparterranno a nessuna chiave

$F^+ = FEG \implies \text{non è chiave}$

$B^+ = BACGDEF \implies \text{chiave}$

$C^+ = CFBEGAD \implies \text{chiave}$

$A^+ = ACGDEFB \implies \text{chiave}$

E2.2-Controllare se F è in 3NF

Dato uno schema R e un insieme di dipendenze funzionali F, per sapere se è in 3NF devo per ogni dipendenza $(X \to Y)$ controllare se è scritta in 3NF, cioè:

- X è superchiave o chiave
- Y è primo, cioè contenuto in una chiave

Esempio:

$R = ABCDEFG$

$F = \{AC \to DE, C \to FB, BC \to EG, B \to A, A \to CG, B \to C, F \to EG\}$

$K = \{A, B, C\}$

Controllo le dipendenze:

- $AC \to DE \implies \text{va bene perchè } AC \supseteq A$
- $C \to FB \implies \text{va bene perchè } C \in K$
- $BC \to EG \implies \text{va bene perchè } BC \supseteq B$
- $B \to A \implies \text{va bene perchè } B \in K$
- $A \to CG \implies \text{va bene perchè } A \in K$
- $B \to C \implies \text{va bene perchè } B \in K$
- $F \to EG \implies \text{non va bene perchè } F \not\supseteq K \text{ e } EG \not\subseteq K$

E2.3-Controllare se una decomposizione preserva F

Data uno schema relazionale R e una decomposizione $\rho = R_1, ..., R_n$

Per ogni dipendenza $(X \to Y)$ devo eseguire dei passi ciclici:

1. Calcolo per ogni sottoschema R_i e faccio l'unione:

$$Z_i = ((X \cap R_1)^+ \cap R_1) \cup ... \cup ((X \cap R_n)^+ \cap R_n)$$

2. Nel caso in cui $Y \subseteq Z_i$ posso fermarmi e sapere che la dipendenza è preservata, invece se $Z_i \not\subseteq X$ rifaccio il passo 1 usando come $X = Z_i$
3. Nel caso in cui $Z_i \subseteq X$ e $Y \not\subseteq Z_i$ la dipendenza non è preservata

Appena trovo una dipendenza non preservata posso fermarmi e sapere che F non è preservata, invece se tutte le dipendenza sono preservate allora F è preservata

Esempio:

$R = ABCDE$

$F = \{AB \to C, AB \to D, B \to E, CD \to E\}$

$\rho = \{ABC, CDE\}$

Controllo le dipendenze:

- $AB \to C$:

  $$Z_1 = ((AB \cap ABC)^+ \cap ABC) \cup ((AB \cap CDE)^+ \cap CDE) = (AB^+ \cap ABC) \cup \emptyset = ABC \implies C \in ABC \implies \text{preservata}$$

- $AB \to D$:

  $$Z_1 = ((AB \cap ABC)^+ \cap ABC) \cup ((AB \cap CDE)^+ \cap CDE) = (AB^+ \cap ABC) \cup \emptyset = ABC \implies ABC \not\subseteq AB$$

  $$Z_2 = ((ABC \cap ABC)^+ \cap ABC) \cup ((ABC \cap CDE)^+ \cap CDE) = (ABC^+ \cap ABC) \cup (C^+ \cap CDE) = ABC \cup C = ABC \implies ABC \subseteq ABC$$

  $$D \notin ABC \implies \text{non preservata}$$

F non è preservata

E2.4-Controllare se una decomposizione ha un join senza perdita

Data uno schema relazionale R e una decomposizione ρ

1. Si costruisce una tabella con gli attributi come colonne e le decomposizioni come righe
2. In ogni casella della tabella con colonna j e riga i scriviamo $\begin{cases} a & A_j \in R_i \\ b_i & A_j \notin R_i \end{cases}$
3. Per ogni dipendenza $(X \to Y) \in F$ se ci sono due tuple $t_1[X] = t_2[X]$ ma $t_1[Y] \neq t_2[Y]$ allora se una delle due è uguale ad a cambio anche l'altra in a mentre se nessuno dei due è uguale a a allora cambio uno delle due facendola diventare uguale all'altra.
4. Se in qualunque momento una riga avesse tutte a allora possiamo fermarci e sapere che ha un join senza perdite
5. Alla fine del controllo di tutte le dipendenze se nessuna riga ha tutte a allora ripartiamo a controllare su tutte le dipendenze dal punto 3, fino a quando un'intero ciclo di dipendenze non cambia nulla sulla tabella
6. Se dopo un ciclo in cui non viene cambiato nulla nella tabella nessuna riga ha tutte a vuol dire che la decomposizione non ha un join senza perdita

Esempio:

$R = ABCDEG$

$F = \{G \to AB, A \to E, E \to B, BE \to G\}$

$\rho = \{ABC, ABE, CDG\}$

Tabella:

	A	B	C	D	E	G
ACD	a	$b_1 \xrightarrow{1} a$	a	a	$b_1 \xrightarrow{1} a$	$b_1 \xrightarrow{1} b_2$
ABE	a	a	b_2	b_2	a	b_2
CDG	b_3	b_3	a	a	b_3	a

1. Ciclo 1
   - $G \rightarrow AB$
   - $A \rightarrow E \implies t_1[E] = a$
   - $E \rightarrow B \implies t_1[B] = a$
   - $BE \rightarrow G \implies t_1[G] = b_2$
2. Ciclo 2
   - $G \rightarrow AB$
   - $A \rightarrow E$
   - $E \rightarrow B$
   - $BE \rightarrow G$

Nessuna riga contiene solo a quindi ρ non ha un join senza perdita.

E2.5-Trovare una copertura minimale e una decomposizione ottimale

Dato uno schema R per trovare la copertura minimale bisogna seguire 3 passaggi:

1. Divido tutte le dipendenze che hanno più di un attributo determinato e le trasformo in diverse dipendenze con solo un attributo determinato: $X \rightarrow YZ \implies \begin{cases} X \rightarrow Y \\ X \rightarrow Z \end{cases}$
2. Per ogni dipendenza devo controllare se un qualche sottoinsieme del determinante contiene il determinato, nel caso sostituisco questa nuova dipendenza alla vecchia:

   $$\exists Z \subset X | Y \in (Z)^+ \implies \begin{cases} \cancel{(X \rightarrow Y)} \\ (Z \rightarrow Y) \end{cases}$$

3. Per ogni dipendenza calcolo X^+ senza però contare la dipendenza, se Y è comunque contenuto nel risultato allora posso eliminare la dipendenza:

$$Y \in (X)^+_{\{F \setminus \{X \to Y\}\}} \implies \cancel{(X \to Y)}$$

Dopo aver trovato la copertura minimale per trovare la decomposizione ottimale bisogna seguire dei passaggi:

1. Dobbiamo inserire tutti gli attributi che non compaiono in nessuna dipendenza come sottoschema
2. Se dopo questo esiste una dipendenza che comprende tutti gli attributi rimanenti li aggiungo come sottoschema
3. Se non c'è per ogni dipendenza aggiungo tutti gli attributi che appaiono in quella determinata dipendenza come sottoschema
4. Nel caso in cui alla fine non ci fosse nessun sottoschema che comprenda una chiave andrebbe aggiunta una qualsiasi chiave come sottoschema per avere un join senza perdita

Esempio:

$$R = ABCDEFG$$

$$F = \{AC \to DE, C \to FB, BC \to EG, B \to A, A \to CG, B \to C, F \to EG\}$$

1. $$F = \{AC \to D, AC \to E, C \to F, C \to B, BC \to E, BC \to G,$$
$$B \to A, A \to C, A \to G, B \to C, F \to E, F \to G\}$$

2. $$AC \to D \implies C^+ = ABCDEFG \implies \cancel{AC \to D}$$
$$AC \to E \implies C^+ = ABCDEFG \implies \cancel{AC \to E}$$
$$BC \to E \implies C^+ = ABCDEFG \implies \cancel{BC \to E}$$
$$BC \to G \implies C^+ = ABCDEFG \implies \cancel{BC \to G}$$

$$F = \{C \to D, C \to E, C \to F, C \to B, C \to G, B \to A, A \to C,$$
$$A \to G, B \to C, F \to E, F \to G\}$$

3. $$C \to D \implies (C)^+_{F \setminus \{C \to D\}} = ABCEFG$$
$$C \to E \implies (C)^+_{F \setminus \{C \to E\}} = ABCDEFG \implies \cancel{C \to E}$$
$$C \to F \implies (C)^+_{F \setminus \{C \to F\}} = ABCDEG$$
$$C \to B \implies (C)^+_{F \setminus \{C \to B\}} = CDEFG$$
$$C \to G \implies (C)^+_{F \setminus \{C \to G\}} = ABCDEFG \implies \cancel{C \to G}$$
$$B \to A \implies (B)^+_{F \setminus \{B \to A\}} = BCDEFG$$
$$A \to C \implies (A)^+_{F \setminus \{A \to C\}} = AG$$

$$A \to G \implies (A)^+_{F\setminus\{A\to G\}} = ABCDEFG \implies \cancel{A \to G}$$

$$B \to C \implies (B)^+_{F\setminus\{B\to C\}} = ABCDEFG \implies \cancel{B \to C}$$

$$F \to E \implies (F)^+_{F\setminus\{F\to E\}} = FG$$

$$F \to G \implies (F)^+_{F\setminus\{F\to G\}} = FE$$

$F = \{C \to D, C \to F, C \to B, B \to A, A \to C, F \to E, F \to G\}$

Tutti gli attributi appaiono in qualche dipendenza e nessuna dipendenza li comprende tutti quindi aggiungo un sottoschema per ogni dipendenza:

$\rho = \{CD, CF, CB, BA, AC, FE, FG\}$

C'è almeno un sottoschema che contiene una chiave quindi ha un join senza perdita.

E3-Esercizi sull'organizzazione fisica

E3.1-Tempi di ricerca sequenziale e random

E3.1.1-Notazione

Dato	**Significato**	**Unità di misura**
Seek	Tempo impiegato per spostare l'attuatore	ms
VelA	Velocità di rotazione dell'asse	rpm
ROT	Tempo necessario per mettere il disco in posizione	ms
BlS	Grandezza del blocco	Byte
TR	Velocità di trasferimento dati	MBps
T_{RBA}	Tempo per effettuare un accesso random	ms
T_{SBA}	Tempo per effettuare un accesso sequenziale	ms

E3.1.2-Formule

Dato	Formula
ROT	$\frac{60000}{\text{VelA}}$
T_{RBA}	$\text{Seek} + \frac{\text{ROT}}{2} + \frac{\text{BlS}}{\text{TR}\cdot 2^{20}} \cdot 10^3$
T_{SBA}	$\frac{\text{ROT}}{2} + \frac{\text{BlS}}{\text{TR}\cdot 2^{20}} \cdot 10^3$

Esempio:

Seek	VelA	TR	BlS
8.9 ms	7200 rpm	150 MBps	4096 Byte

$$\text{ROT} = \frac{60000}{7200} = 8.33 \text{ ms}$$

$$T_{\text{RBA}} = 8.9 + \frac{8.33}{2} + \frac{4096}{150\cdot 2^{20}} \cdot 10^3 = 13.09 \text{ ms}$$

$$T_{\text{SBA}} = \frac{8.33}{2} + \frac{4096}{150\cdot 2^{20}} \cdot 10^3 = 4.19 \text{ ms}$$

E3.2-Organizzazione tramite file heap e sequenziali

E3.2.1-Notazione

Dato	Significato	Unità di misura
NR	Numero di record	
RS	Grandezza di un record	Byte
RpBl	Numero di record per blocco	
NBl	Numero di blocchi	
NA_{RBA}	Numero di accessi usando RBA	
NA_{SBA}	Numero di accessi usando SBA	

E3.2.2-Formule

Dato	Formula
RpBl	$\lfloor \frac{\text{BlS}}{\text{RS}} \rfloor$
NBl	$\lceil \frac{\text{NR}}{\text{RpBl}} \rceil$
NA_{SBA}	$\lceil \frac{\text{NBl}}{2} \rceil$
NA_{RBA}	$\lceil \log_2(\text{NBl}) \rceil$

Esempio:

NR	BlS	RS
30000	2048 Byte	100 Byte

$\text{RpBl} = \lfloor \frac{2048}{100} \rfloor = 20$

$\text{NBl} = \lceil \frac{30000}{20} \rceil = 1500$

$\text{NA}_{\text{SBA}} = \lceil \frac{1500}{2} \rceil = 750 \text{ SBA}$

$\text{NA}_{\text{RBA}} = \lceil \log_2(1500) \rceil = 11 \text{ RBA}$

E3.3-Organizzazione tramite file hash

E3.3.1-Notazione

Dato	Significato	Unità di misura
PS	Grandezza del puntatore	Byte
NBk	Numero di bucket	
PpBl	Numero di puntatori per blocco	
BlpBD	Numero di blocchi necessari per la bucket directory	
RpBl	Numero di record per blocco	
RpBk	Numero di record per bucket	
BlpBk	Numero di blocchi per ogni bucket	
Bl_{Bk}	Numero di blocchi per tutti i bucket	
Bl_{TOT}	Blocchi necessari per contenere tutto	
NA	Numero di accessi per effettuare una ricerca	

E3.3.2-Formule

Dato	Formula
PpBl	$\lfloor \frac{\text{BlS}}{\text{PS}} \rfloor$
BlpBD	$\lceil \frac{\text{NBk}}{\text{PpBl}} \rceil$
RpBl	$\lfloor \frac{\text{BlS}-\text{PS}}{\text{RS}} \rfloor$
RpBk	$\lceil \frac{\text{NR}}{\text{NBk}} \rceil$
BlpBk	$\lceil \frac{\text{RpBk}}{\text{RpBl}} \rceil$
Bl_{Bk}	$\text{BlpBk} \cdot \text{NBk}$
Bl_{TOT}	$\text{Bl}_{\text{Bk}} + \text{BlpBD}$
NA	$\lceil \frac{\text{BlpBk}}{2} \rceil$

Esempio:

NR	BlS	RS	PS	NBk
1857000	4096 Byte	256 Byte	5 Byte	300

$\text{PpBl} = \lfloor \frac{4096}{5} \rfloor = 819$

$\text{BlpBD} = \lceil \frac{300}{819} \rceil = 1$

$\text{RpBl} = \lfloor \frac{4096-5}{256} \rfloor = 15$

$\text{RpBk} = \lceil \frac{1857000}{300} \rceil = 6190$

$\text{BlpBk} = \lceil \frac{6190}{15} \rceil = 413$

$\text{Bl}_{\text{Bk}} = 300 \cdot 413 = 123900$

$\text{Bl}_{\text{TOT}} = 123900 + 1 = 123901$

$\text{NA} = \lceil \frac{413}{2} \rceil = 206$

E3.4-Organizzazione tramite ISAM

E3.4.1-Notazione

Dato	Significato	Unità di misura
KS	Grandezza della chiave	Byte
BlpFP	Numero di blocchi necessari per il file principale	
IS	Grandezza di ogni entrata del file indice	Byte
IpBl	Numero di entrate per blocco del file indice	
BlpFI	Numero di blocchi necessari per il file indice	
NA	Numero di accessi per effettuare una ricerca	

E3.4.2-Formule

Dato	Formula
RpBl	$\lfloor \frac{\mathrm{BlS}}{\mathrm{RS}} \rfloor$
BlpFP	$\lceil \frac{\mathrm{NR}}{\mathrm{RpBl}} \rceil$
IS	KS + PS
IpBl	$\lfloor \frac{\mathrm{BlS}}{\mathrm{IS}} \rfloor$
BlpFI	$\lceil \frac{\mathrm{BlpFP}}{\mathrm{IpBl}} \rceil$
$\mathrm{NA_{FI}}$	$\lceil \log_2(\mathrm{BlpFI}) \rceil + 1$
$\mathrm{NA_{FP}}$	$\lceil \log_2(\mathrm{BlpFP}) \rceil$

Esempio:

NR	BlS	RS	PS	KS
1750000	2048 Byte	130 Byte	5 Byte	35 Byte

$\mathrm{RpBl} = \lfloor \frac{2048}{130} \rfloor = 15$

$\mathrm{BlpFP} = \lceil \frac{1750000}{15} \rceil = 116667$

$\mathrm{IS} = 35 + 5 = 40 \text{ Byte}$

$\mathrm{EpBl} = \lfloor \frac{2048}{40} \rfloor = 51$

$\mathrm{BlpFI} = \lceil \frac{116667}{51} \rceil = 2288$

$\mathrm{NA_{FI}} = \lceil \log_2(2288) + 1 \rceil = 13$

E3.5-Organizzazione tramite B-Tree(bilanciato)

E3.5.1-Notazione

Dato	Significato
n	Numero di chiavi totali
m	Massimo numero di figli di un nodo
d	Minimo numero di figli di un nodo
%Bl	Minimo di riempimento di un blocco
NP	Numero di puntatori in un blocco
NK	Numero di chiavi in un blocco
$\mathrm{NB}_{\mathrm{L}i}$	Numero blocchi al livello i

E3.5.2-Formule

Dato	Formula
n_{max}	$\mathrm{m}^{\mathrm{h}} - 1$
n_{min}	$2\mathrm{d}^{\mathrm{h}-1} - 1$
d	$\lceil \frac{\mathrm{m}}{2} \rceil$
$\mathrm{h}_{min}(\%\mathrm{Bl} = 50\%)$	$\lceil \log_m(\mathrm{n}_{min} + 1) \rceil$
$\mathrm{h}_{max}(\%\mathrm{Bl} = 100\%)$	$\lfloor \log_d(\frac{\mathrm{n}_{max}+1}{2}) \rfloor + 1$
RpBl	$\lfloor \frac{\mathrm{BlS} \cdot \%\mathrm{Bl}}{\mathrm{RS}} \rfloor$
BlpFP	$\lceil \frac{\mathrm{NR}}{\mathrm{RpBl}} \rceil$
NK	$\lceil \frac{\mathrm{BlS} \cdot \%\mathrm{Bl} - \mathrm{PS}}{\mathrm{KS} + \mathrm{PS}} \rceil$
NP	$\mathrm{NK} + 1$
$\mathrm{Bl}_{\mathrm{L}1}$	$\lceil \frac{\mathrm{BlpFP}}{\mathrm{NP}} \rceil$
$\mathrm{Bl}_{\mathrm{L}(i+1)}$	$\lceil \frac{\mathrm{Bl}_{\mathrm{L}i}}{\mathrm{NP}} \rceil$
BlpFI	$\sum_{i=1} \mathrm{Bl}_{\mathrm{L}i}$
NA	$O(\mathrm{h})$

Esempio:

NR	BlS	RS	PS	KS
1750000	2048 Byte	250 Byte	4 Byte	45 Byte

$$\mathrm{RpBl} = \lfloor \frac{2048}{250} \rfloor = 8$$

$\text{BlpFP} = \lceil \frac{1750000}{8} \rceil = 218750$

$\text{NK} = \lceil \frac{2048-4}{45+4} \rceil = 42$

$\text{NP} = 42 + 1 = 43$

$\text{Bl}_{\text{L1}} = \lceil \frac{218750}{43} \rceil = 5088$

$\text{Bl}_{\text{L2}} = \lceil \frac{5088}{43} \rceil = 119$

$\text{Bl}_{\text{L3}} = \lceil \frac{119}{43} \rceil = 3$

$\text{Bl}_{\text{L4}} = \lceil \frac{3}{43} \rceil = 1$

$\text{BlpFI} = 5088 + 119 + 3 + 1 = 5211$

$\text{h} = 4$

$\text{d} = \lceil \frac{43}{2} \rceil = 22$

$\text{n} = 43^{5+1} = 6321363049$