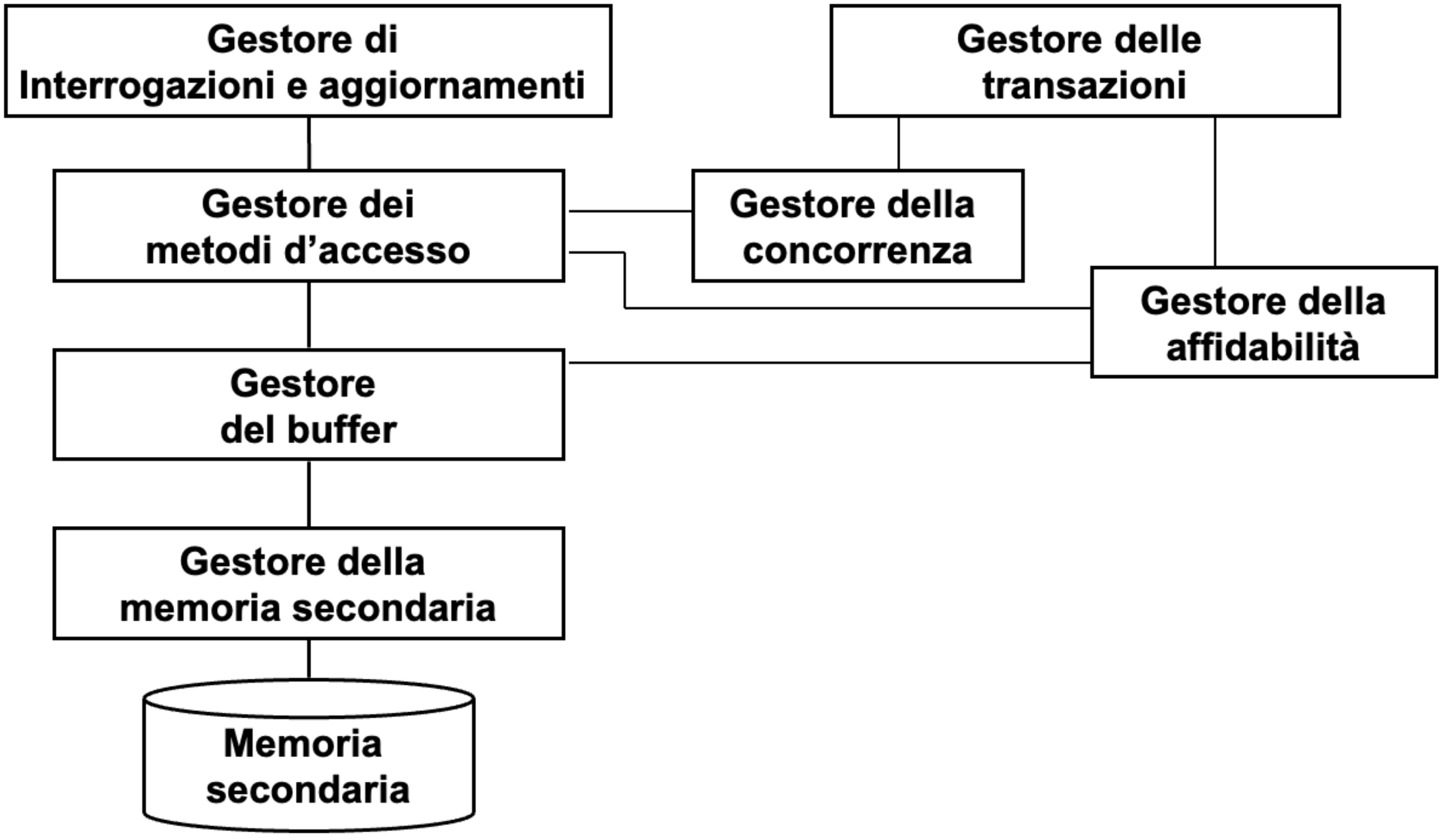
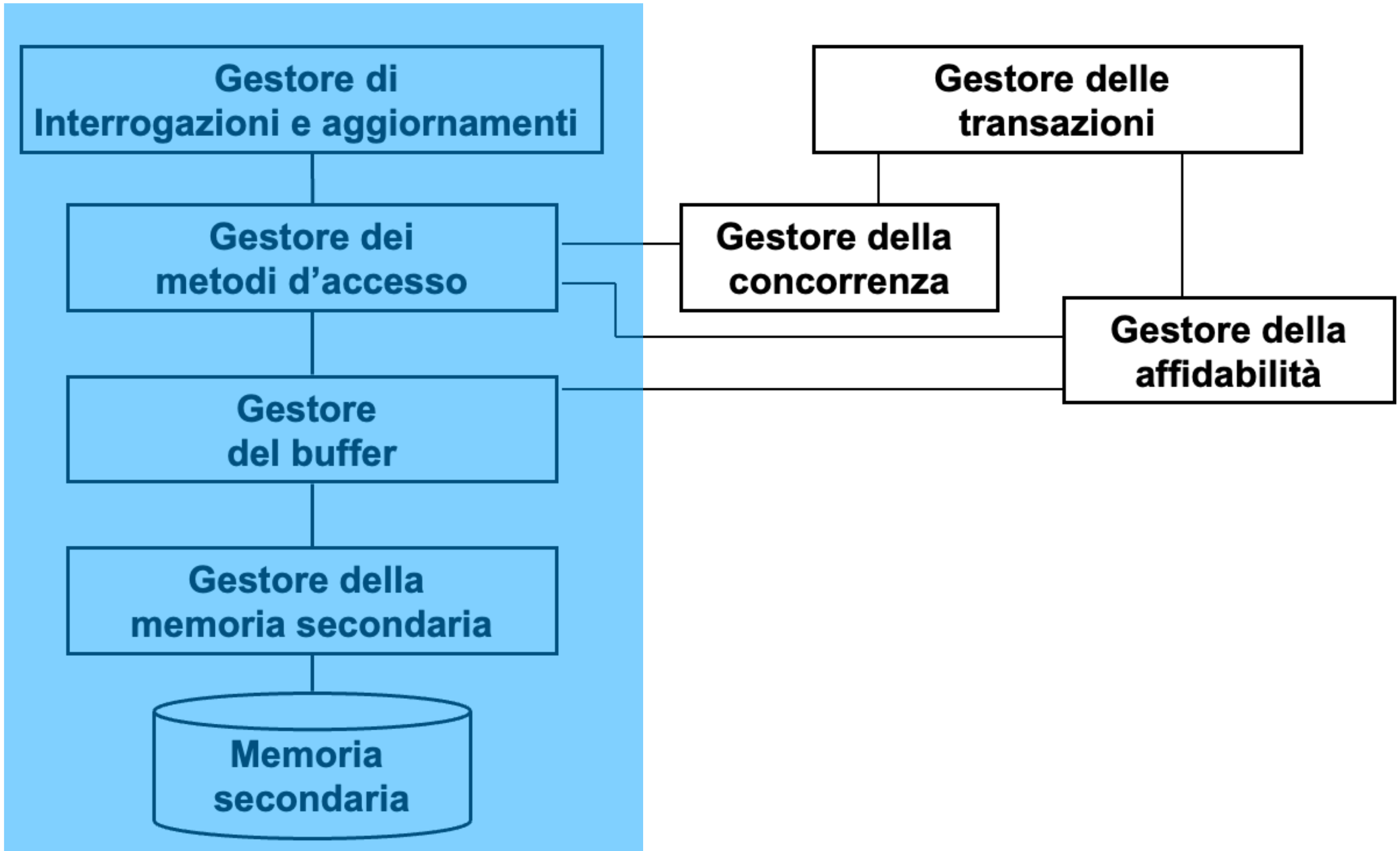
Gestione delle interrogazioni e degli aggiornamenti


Gestore di
Interrogazioni e aggiornamenti
Gestore delle
transazioni
Gestore dei
metodi d'accesso
Gestore della
concorrenza
Gestore della
affidabilità
Gestore
del buffer
Gestore della
memoria secondaria
Memoria
secondaria


Gestore di
Interrogazioni e aggiornamenti
Gestore dei
metodi d'accesso
Gestore
del buffer
Gestore della
memoria secondaria
Memoria
secondaria
Gestore delle
transazioni
Gestore della
concorrenza
Gestore della
affidabilità

Memoria Principale e Secondaria

- I **programmi** possono fare riferimento solo a **dati** in **memoria principale**
- Le **basi di dati** debbono essere (sostanzialmente) in **memoria secondaria** per due motivi:
  - dimensioni
  - persistenza
- I **dati** in **memoria secondaria** possono essere utilizzati solo se prima trasferiti in **memoria principale**
  - questo spiega i termini “principale” e “secondaria”

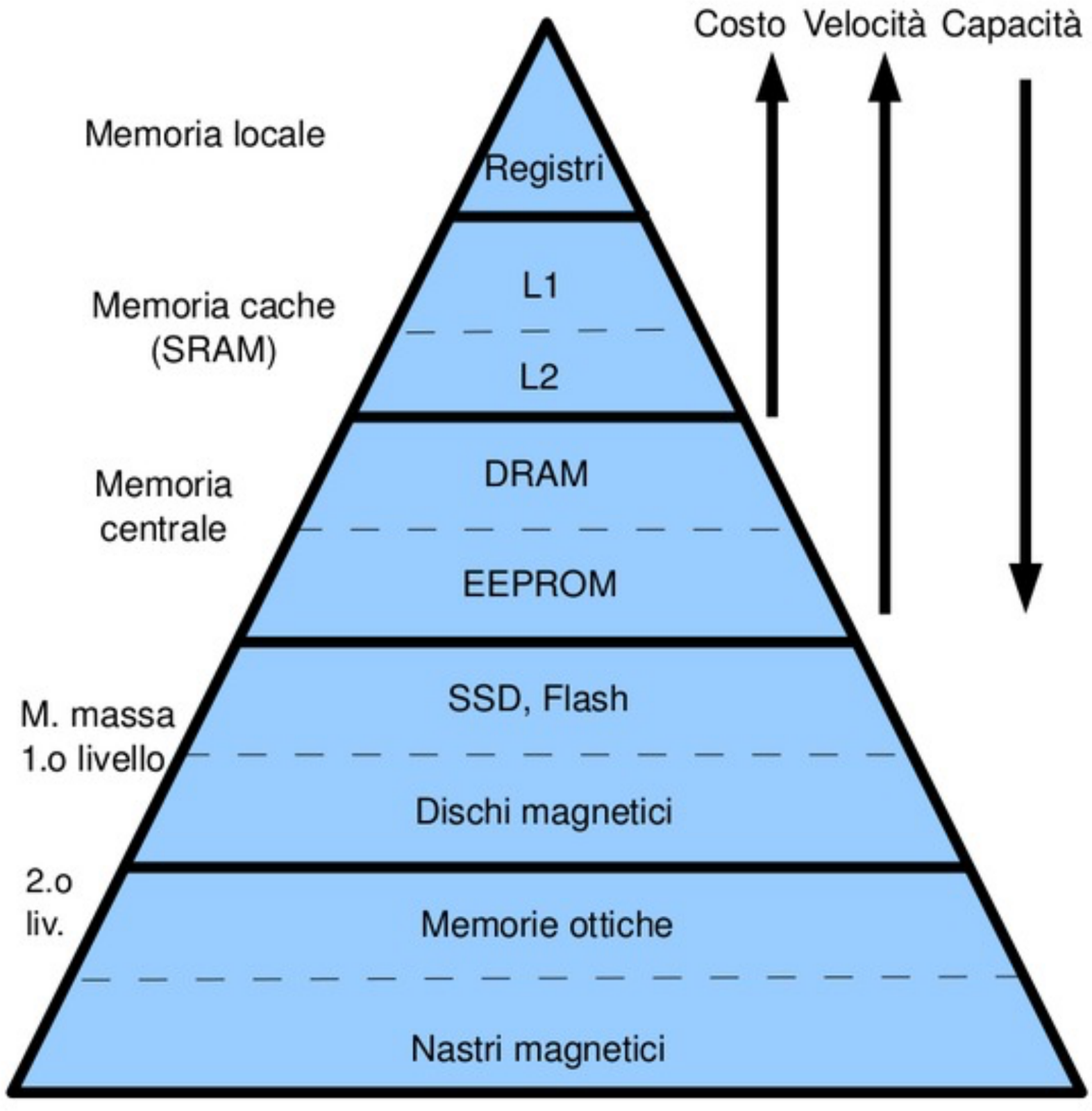
Memoria Principale e Secondaria

- I dispositivi di **memoria secondaria** sono organizzati in **blocchi** di **lunghezza** (di solito) **fissa** (ordine di grandezza: alcuni KB)
- Le **uniche operazioni** sui dispositivi solo la **lettura** e la **scrittura** di di una **pagina**, cioè dei **dati di un blocco** (cioè di una stringa di byte);
  - Per comodità consideriamo **blocco** e **pagina sinonimi**
- Il ***filesystem*** è il componente del **sistema operativo** che **gestisce la memoria secondaria**
- I DBMS ne utilizzano le funzionalità per **creare** ed **eliminare file** e per **leggere** e **scrivere singoli blocchi** o **sequenze di blocchi contigui**

Memoria Principale e Secondaria

- Il DBMS gestisce i **file allocati** come se fossero un **unico grande spazio di memoria secondaria** e costruisce, in tale spazio, le **strutture fisiche** con cui implementa le relazioni
- **L'organizzazione dei file**, sia in termini di distribuzione dei record nei blocchi sia relativamente alla struttura all'interno dei singoli blocchi, **è gestita direttamente dal DBMS**
- Il DBMS crea **file** di **grandi** dimensioni che utilizza per **memorizzare** diverse **relazioni** (al limite, l'intero database)
  - È possibile che un file contenga i dati di più relazioni e che le varie n-uple di una relazione siano in file diversi
  - Spesso, ma non sempre, ogni blocco è dedicato a n-uple di un'unica relazione

Gerarchia di memoria



Memoria Secondaria

- Dato un **indirizzo di accesso**, le **prestazioni** di memoria secondaria si misurano in termini della **somma** tra
  - il **tempo** che la testina impiega per **raggiungere** la **traccia** di interesse,
  - la **latenza** (tempo per accedere al primo byte del blocco di interesse)
  - il **tempo di trasferimento** (tempo necessario a muovere tutti i dati del blocco)

Memoria Secondaria

- **Accesso a memoria secondaria**:
  - tempo di posizionamento della testina (10 - 50 ms)
  - tempo di latenza (5 - 10 ms)
  - tempo di trasferimento (1 - 2 ms)
- In **media** non meno di **10 ms**
- Il costo di un accesso a memoria secondaria è **quattro o più ordini di grandezza maggiore** di quello per operazioni in memoria centrale
- Perciò, nelle applicazioni "**I/O bound**" (cioè con molti accessi a memoria secondaria e relativamente poche operazioni) il **costo** dipende esclusivamente dal **numero di accessi** a memoria secondaria
- Inoltre, accessi a **blocchi "vicini" costano meno** (contiguità)

Gestione del buffer

- **Buffer**:
  - area di **memoria centrale**, **gestita** dal DBMS (preallocata) e **condivisa** fra le transazioni
  - **organizzato in pagine** di dimensioni pari o multiple di quelle dei **blocchi di memoria** secondaria (1 KB - 100 KB)
- Se assumiamo che **coincidano pagina e blocco**, il caricamento di **una pagina** del buffer richiede **una lettura** in memoria secondaria, mentre salvare una pagina corrisponde ad **una scrittura**

Scopo della gestione del buffer

- **Ridurre il numero di accessi** alla memoria secondaria
  - In caso di **lettura**, se la pagina è già presente nel buffer, non è necessario accedere alla memoria secondaria
  - In caso di **scrittura**, il gestore del buffer può decidere di differire la scrittura fisica (ammesso che ciò sia compatibile con la gestione dell'**affidabilità**)
- La **gestione dei buffer** e la **differenza di costi** fra memoria principale e secondaria possono suggerire **algoritmi innovativi**
- Esempio:
  - File di 10.000.000 di record di 100 byte ciascuno (1GB)
  - Blocchi di 4KB
  - Buffer disponibile di 20M
- Come possiamo fare **l'ordinamento**?
  - **Merge-sort "a più vie"** (*multi-way mergesort*)

Dati gestiti dal buffer manager

- Il **buffer** stesso
- Una **directory** che **per ogni pagina** mantiene (ad esempio)
  - il **file fisico** e il **numero del blocco**
- **due variabili di stato**:
  - un **contatore** che indica quanti programmi utilizzano la pagina
  - un **bit** che indica se la pagina è "sporca", cioè se è stata modificata

Funzioni del buffer manager

- *Intuitivamente*:
  - riceve **richieste** di **lettura** e **scrittura** (di pagine)
  - le **esegue** accedendo alla **memoria secondaria** solo quando **indispensabile** e utilizzando invece il **buffer** quando **possibile**
  - esegue le **primitive** `fix`, `unfix`, `setDirty`, `force`
- Le **politiche** sono simili a quelle relative alla gestione della memoria da parte dei sistemi operativi:
  - "**località dei dati**": è alta la probabilità di dover riutilizzare i dati attualmente in uso
  - "**legge 80-20**": l'80% delle operazioni utilizza sempre lo stesso 20% dei dati

Blocchi e n-uple

- I **file** sono **logicamente** organizzati in **record**
- I **record** sono mappati nei **blocchi** di memoria secondaria
- Le n-**uple** di una relazione (record di file) stanno in **blocchi contigui**
- A volte in un blocco ci sono n-**uple di relazioni diverse ma correlate** (i join sono favoriti)
- I **blocchi** (componenti "**fisici**" di un **file**) e le n-**uple** o **record** (componenti "**logici**" di una relazione) hanno dimensioni in generale diverse:
  - la **dimensione del blocco** dipende dal ***file system***
  - la **dimensione del record** dipende dalle esigenze dell'**applicazione**, e può anche variare nell'ambito di un file

Fattore di blocco

- **Numero di record in un blocco**:
  - L_R: **dimensione** di un **record**
    - per semplicità **costante** nel file: "record a lunghezza fissa"
  - L_B: **dimensione** di un **blocco**
  - se $L_B > L_R$, possiamo avere più record in un blocco:

$$\lfloor L_B / L_R \rfloor$$

- Lo **spazio residuo** può essere:
  - **utilizzato** (record "***spanned***")
  - **non utilizzato** (record "***unspanned***")

Esercizio

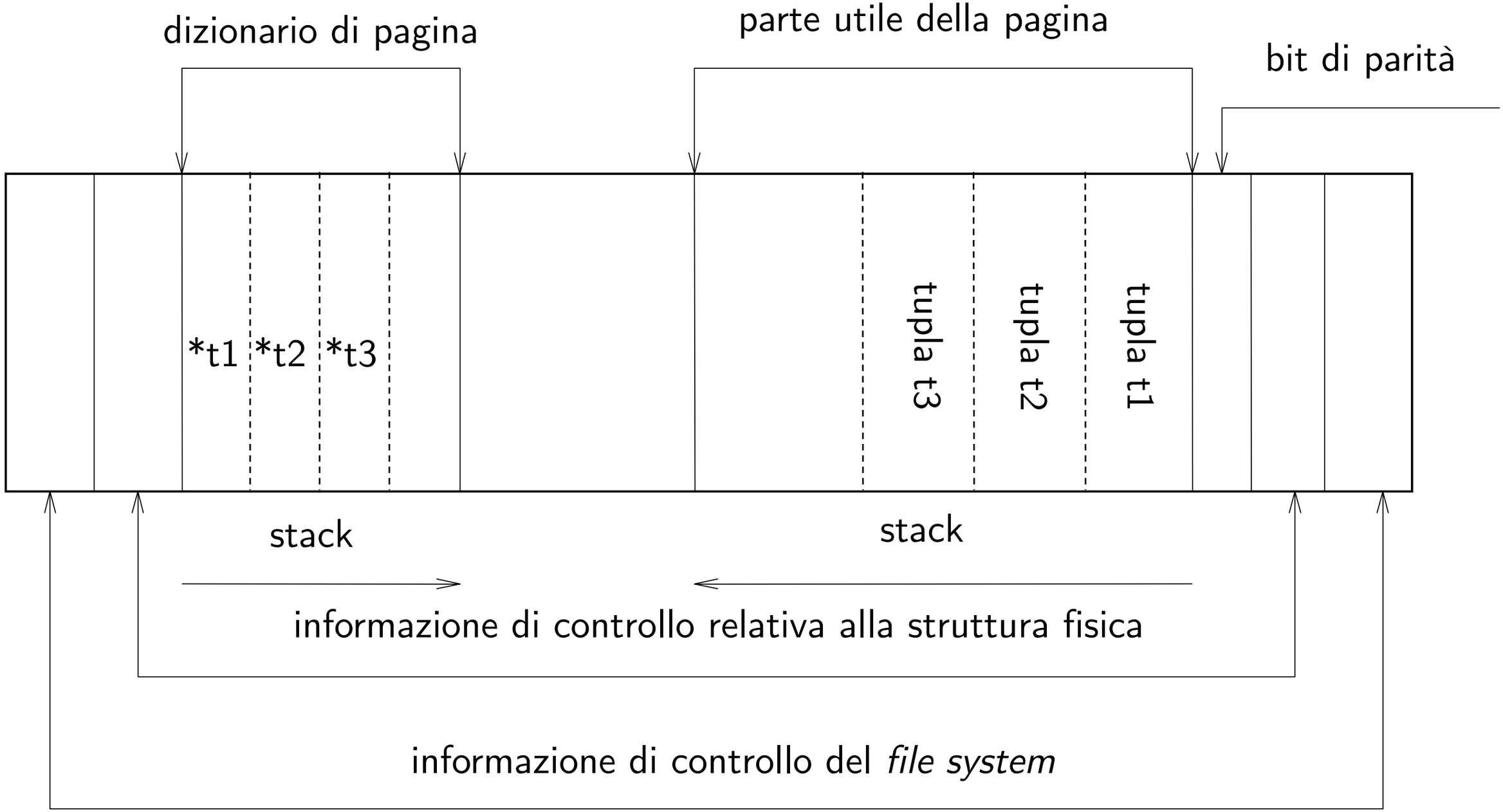
- Calcolare il fattore di blocco e il numero di blocchi occupati da una relazione contenente $T = 512000$ n-uple (record) di lunghezza fissa pari a $L_R = 100$ byte in un sistema con blocchi di dimensione pari a $L_B = 1$ KB

Esercizio

- Calcolare il fattore di blocco e il numero di blocchi occupati da una relazione contenente $T = 512000$ n-uple (record) di lunghezza fissa pari a $L_R = 100$ byte in un sistema con blocchi di dimensione pari a $L_B = 1$ KB

- Fattore di blocco $\lfloor L_B/L_R \rfloor = 10$
- Dimensione totale $D_T = T \times L_R = 512 \times 10^5$ byte
- Numero di blocchi $N_B = \lceil D_T/L_B \rceil = 50000$

Organizzazione delle n-uple nelle pagine



Strutture sequenziali

- Esiste un ordinamento fra le n-uple, che può essere rilevante ai fini della gestione
  - **seriale**: ordinamento fisico ma non logico
  - **ordinata**: l'ordinamento delle tuple coerente con quello di un campo
- **con accesso calcolato**: posizioni individuate attraverso indici

Struttura seriale

- Chiamata anche:
  - *entry-sequenced*
  - file heap
  - file disordinato
- È molto diffusa nelle basi di dati relazionali
- Gli inserimenti vengono effettuati
  - in coda (con riorganizzazioni periodiche)
  - al posto di record cancellati
- La sequenza delle n-uple è indotta dall'ordine di immissione

Struttura ordinata

- Permettono ricerche binarie, ma solo fino ad un certo punto (ad esempio, come troviamo la “metà del file”)?
- Il problema è mantenere l’ordinamento

Struttura con accesso calcolato

- I **file hash** permettono un **accesso calcolato** molto **efficiente**
  - La tecnica si basa su quella utilizzata per le **tavole** (o tabelle) **hash** in memoria centrale

Tavola hash

- Obiettivo: **accesso diretto** ad un insieme di **record** sulla base del **valore di un campo** (detto **chiave**, che per semplicità supponiamo identificante, ma non è necessario)
- Se i **possibili valori** della chiave sono in **numero paragonabile** al **numero di record** (e corrispondono ad un "tipo indice") allora usiamo un array
  - Università con 1000 studenti e numeri di matricola compresi fra 1 e 1000 o poco più e file con tutti gli studenti
- Se i **possibili valori** della chiave sono **molti di più** di quelli **effettivamente utilizzati**, non possiamo usare l'array (spreco);
  - 40 studenti e numero di matricola di 6 cifre (un milione di possibili chiavi)

Tavola hash

- Volendo continuare ad usare qualcosa di **simile ad un array**, ma **senza sprecare spazio**, possiamo pensare di trasformare i valori della chiave in possibili indici di un array
- **Funzione hash**:
  - associa ad ogni **valore** della chiave un "**indirizzo**", in uno spazio di dimensione paragonabile (leggermente superiore) rispetto a quello strettamente necessario
  - poiché il **numero di possibili chiavi** è molto maggiore del **numero di possibili indirizzi** ("lo spazio delle chiavi è più grande dello spazio degli indirizzi"), la funzione **non può essere iniettiva** e quindi esiste la **possibilità di collisioni** (chiavi diverse che corrispondono allo stesso indirizzo)
  - le buone funzioni hash distribuiscono in modo **casuale** e **uniforme**, **riducendo le probabilità di collisione** (che si riduce aumentando lo spazio ridondante)

Esempio

- 40 record
- tavola hash con 50 posizioni:
  - 1 collisione a 4
  - 2 collisioni a 3
  - 5 collisioni a 2
- numero medio di accessi:
  - 32 record x 1 accesso +
  - 5 record x 2 accessi +
  - 2 record x 3 accessi +
  - 1 record x 4 accessi
  - 52 accessi / 40 record = 1.3 accessi in media

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

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

Gestione delle collisioni

- Varie tecniche:
  - **posizioni successive** disponibili
  - tabella di **overflow** (gestita in forma collegata)
  - funzioni **hash "alternative"**
- Notare che:
  - le collisioni ci sono (quasi) sempre
  - le collisioni multiple hanno probabilità che decresce al crescere della molteplicità
  - la molteplicità media delle collisioni è molto bassa

File hash

- L'idea è la stessa, ma si **sfrutta l'organizzazione in blocchi** e il fatto che **l'accesso è al blocco**
- In questo modo si "**ammortizzano**" le **probabilità** di **collisione**

Esempio

- 40 record
- tavola hash con 50 posizioni:
  - 1 collisione a 4
  - 2 collisioni a 3
  - 5 collisioni a 2
- numero medio di accessi:
  - 32 record x 1 accesso +
  - 5 record x 2 accessi +
  - 2 record x 3 accessi +
  - 1 record x 4 accessi
  - 52 accessi / 40 record = 1.3 accessi in media

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

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

Esempio

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

205845

206092

- tavola hash con 50 posizioni:
  - 1 collisione a 4
  - 2 collisioni a 3
  - 5 collisioni a 2
  - numero medio di accessi: 1.3
- file hash con fattore di blocco 10
  - 5 blocchi con 10 posizioni
  - 2 soli overflow
  - numero medio di accessi:
    - $(38 + 4) / 40 = 1.05$

Osservazioni

- È l'organizzazione **più efficiente** per **l'accesso diretto basato su valori della chiave con condizioni di uguaglianza** (accesso puntuale)
  - **costo medio di poco superiore all'unità** (il caso peggiore è molto costoso ma talmente improbabile da poter essere ignorato)
- Le **collisioni** (overflow) sono di solito gestite con **blocchi collegati**
- **Non è efficiente** per **ricerche** basate **su intervalli** (né per ricerche basate su altri attributi)
- I file hash "degenerano" se si riduce lo spazio sovrabbondante: funzionano solo con file la cui dimensione non varia molto nel tempo