

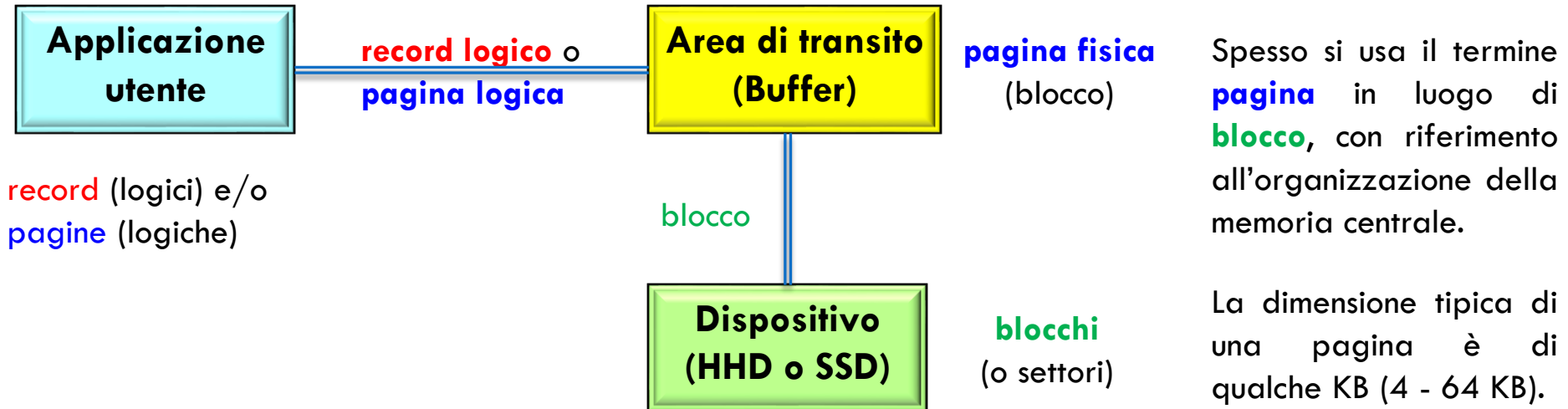
## Livello fisico di un DB

Annalisa Franco, Dario Maio  
Università di Bologna

# DB e dispositivi di memorizzazione

- Un DBMS “convenzionale” gestisce i dati facendo ricorso principalmente a dischi magnetici e solid-state drive (spesso organizzati in configurazioni RAID).
- I dati, per essere elaborati dal DBMS devono essere trasferiti in memoria centrale:
  - il trasferimento non avviene in termini di singole tuple, bensì di **data block** (o **pagine**, termine comunemente usato quando i dati sono in memoria);
  - pagine piccole comportano un maggior numero di operazioni di I/O; pagine grandi tendono ad aumentare la frammentazione interna (pagine parzialmente riempite) e richiedono più spazio in memoria per essere caricate.
- Poiché molto spesso le operazioni di I/O costituiscono il collo di bottiglia del sistema, si rende necessario ottimizzare l'implementazione fisica del DB, attraverso:
  - opportune organizzazioni delle tuple sul/i dispositivo/i fisico/i
  - strutture di accesso efficienti;
  - idonee politiche di gestione dei buffer in memoria;
  - strategie per l'esecuzione per le query.

# Diversi livelli d'astrazione

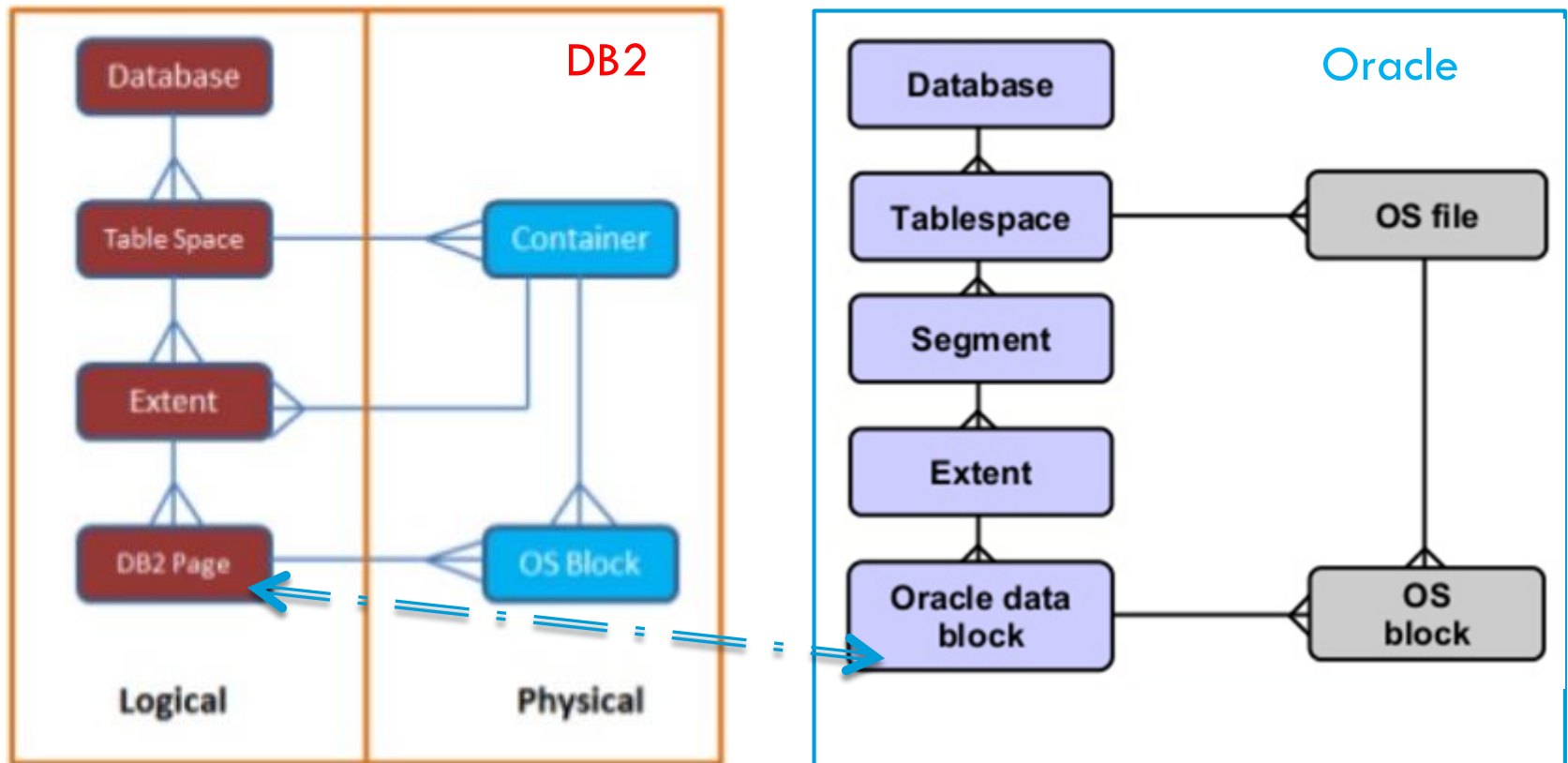


- A livello di **applicazione** si opera normalmente su **record (logici)**.
- A livello di **sistema** di archiviazione si lavora su **blocchi di byte** (es. 4096), la cui dimensione può dipendere dalle caratteristiche del sistema operativo (file system), o può essere stabilita dall'utente. **Un blocco può corrispondere a uno o più record logici, un record logico può occupare più blocchi.**
- A livello di **dispositivo** si lavora ancora su **blocchi di byte**, la cui dimensione può essere scelta dall'utente (nel caso di alcuni dispositivi), o può essere fissa, nel qual caso si parla più propriamente di settori (es. dischi).

# Page vs OS Block

**N.B.** Non tutti i DBMS adottano la definizione di "page". Inoltre, anche in alcuni OS il termine **page** assume importanza per realizzare indipendenza dalla particolare dimensione di block adottata dal dispositivo.

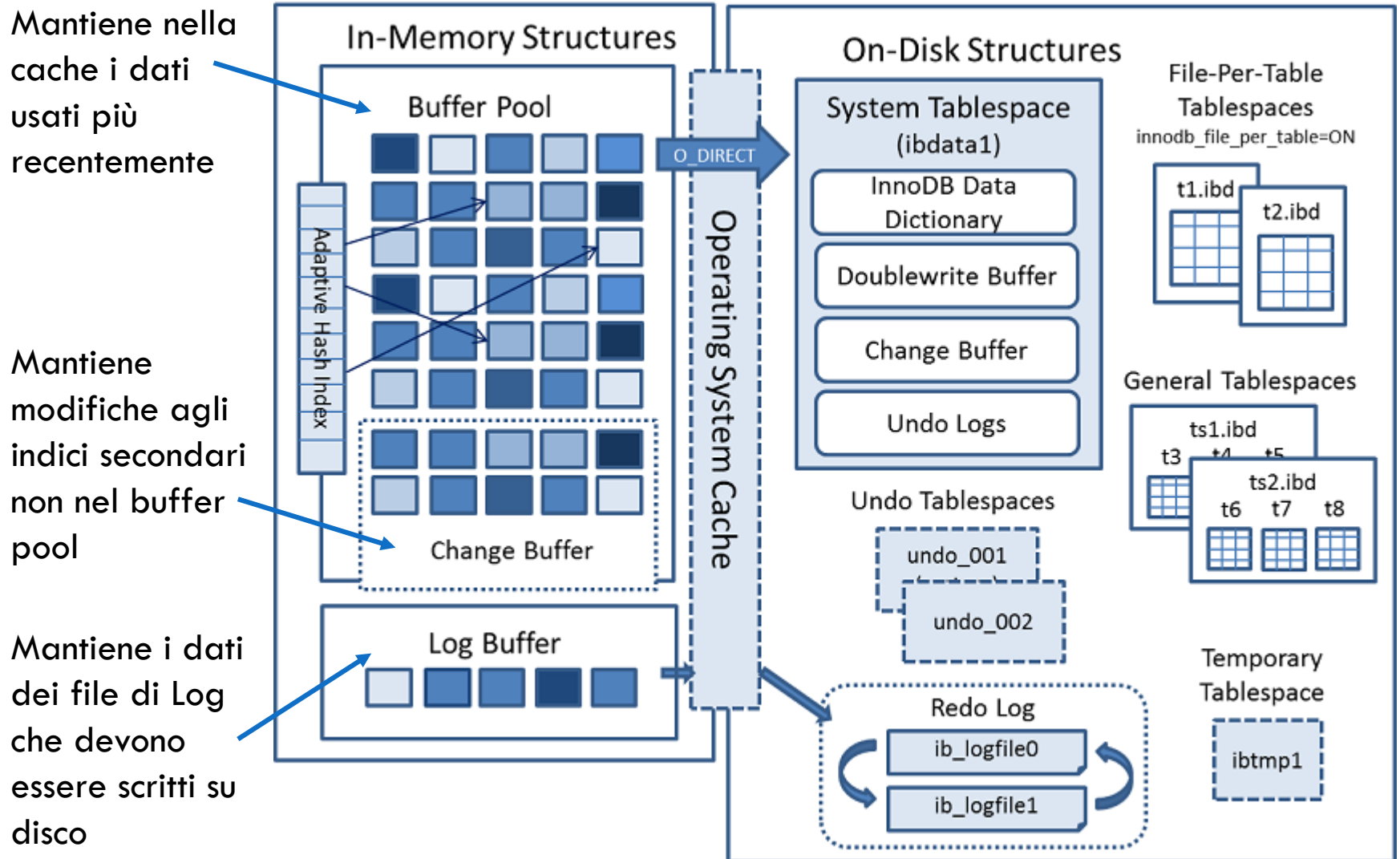
Gerarchia delle strutture logiche e fisiche dei dati in:



# II DB fisico

- A livello fisico un DB consiste di un insieme di file, ognuno dei quali viene visto come una **collezione di pagine**, di dimensione fissa (ad esempio 4 KB).
- Ogni **pagina** memorizza più **record** (corrispondenti alle **tuple logiche**).
- A sua volta un **record** consiste di più **campi**, di lunghezza fissa e/o variabile, che rappresentano gli **attributi**.
- **N.B.** I “file” del DBMS qui considerati non corrispondono necessariamente a quelli del file system del sistema operativo.
- Casi limite:
  - ogni relazione (o tabella) del DB è memorizzata in un proprio file;
  - tutto il DB è memorizzato in un singolo file.
- Ogni DBMS a livello fisico adotta soluzioni specifiche più articolate e flessibili.

# Architettura InnoDB



# Il modello di memorizzazione in MySQL

- Le tabelle definite dagli utenti e i rispettivi indici in InnoDB sono memorizzati in file con estensione .ibd.
- Lo spazio fisico è organizzato in **Tablespaces** che possono essere di due tipi:
  - ▣ **General** (o shared): un singolo file contiene dati appartenenti a diverse tabelle e/o indici;
  - ▣ **File-per-table**: ogni file contiene solo i dati di una tabella e dei suoi indici.

<https://dev.mysql.com/blog-archive/innoDB-tablespace-space-management/>

# Il modello di memorizzazione in MySQL

Un Tablespace in MySQL è una collezione di **extent**.

Un extent è una collezione di pagine contigue all'interno di un Tablespace (dimensione tipica 1 MB)

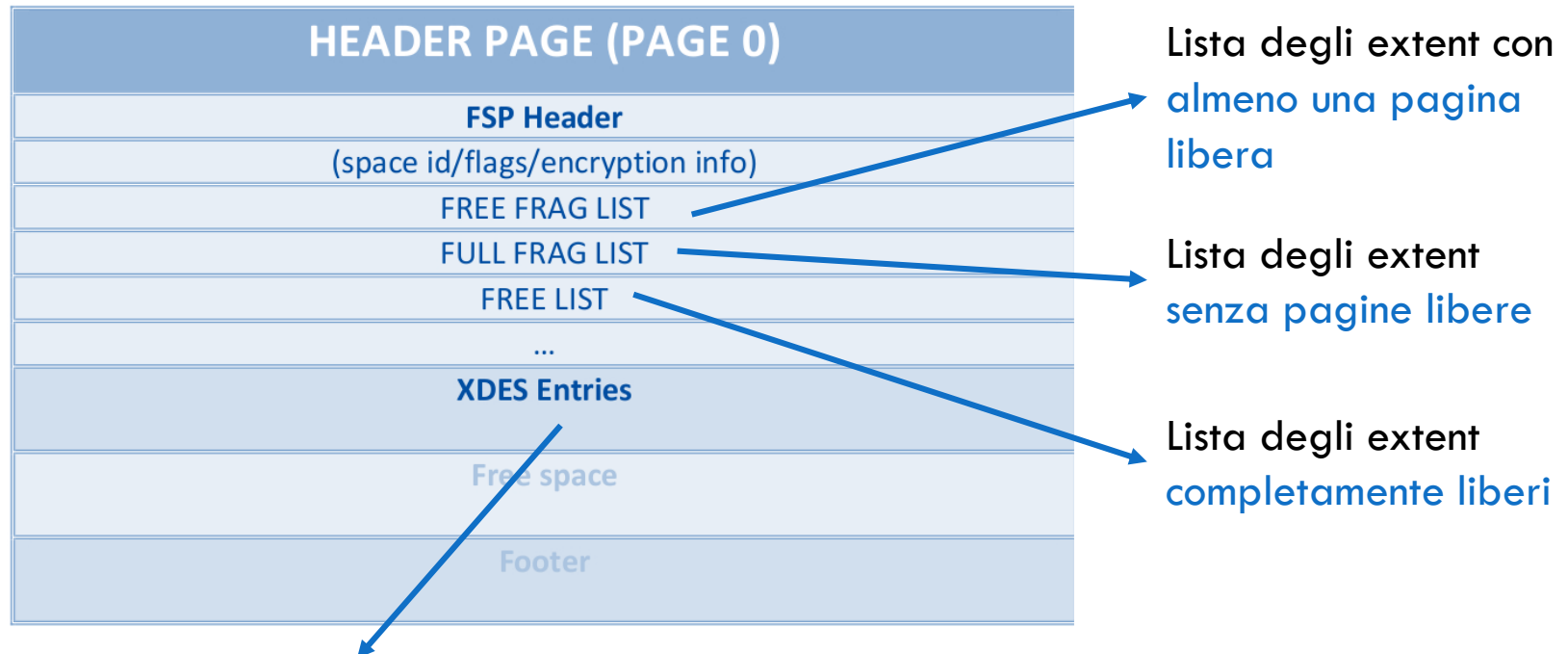
Un insieme di extent è detto **segment**.

TABLESPACE FILE	
Extent 1	Extent 1 Page 0
	Extent 1 Page 1
	...
	Extent 1 Page 63
Extent 2	
...	...
Extent N	



# Il modello di memorizzazione in MySQL

I **metadati** di ciascuno Tablespace sono memorizzati nell'**header** del Tablespace stesso.



**Extent descriptors**, contenenti informazioni su ciascun extent come ad esempio:

- ID del segmento a cui appartiene
- Posizione dell'extent precedente all'interno del segmento
- Posizione dell'extent successivo all'interno del segmento
- Stato dell'extent
- Mappa (bitmap) relativa all'allocazione delle pagine dell'extent.

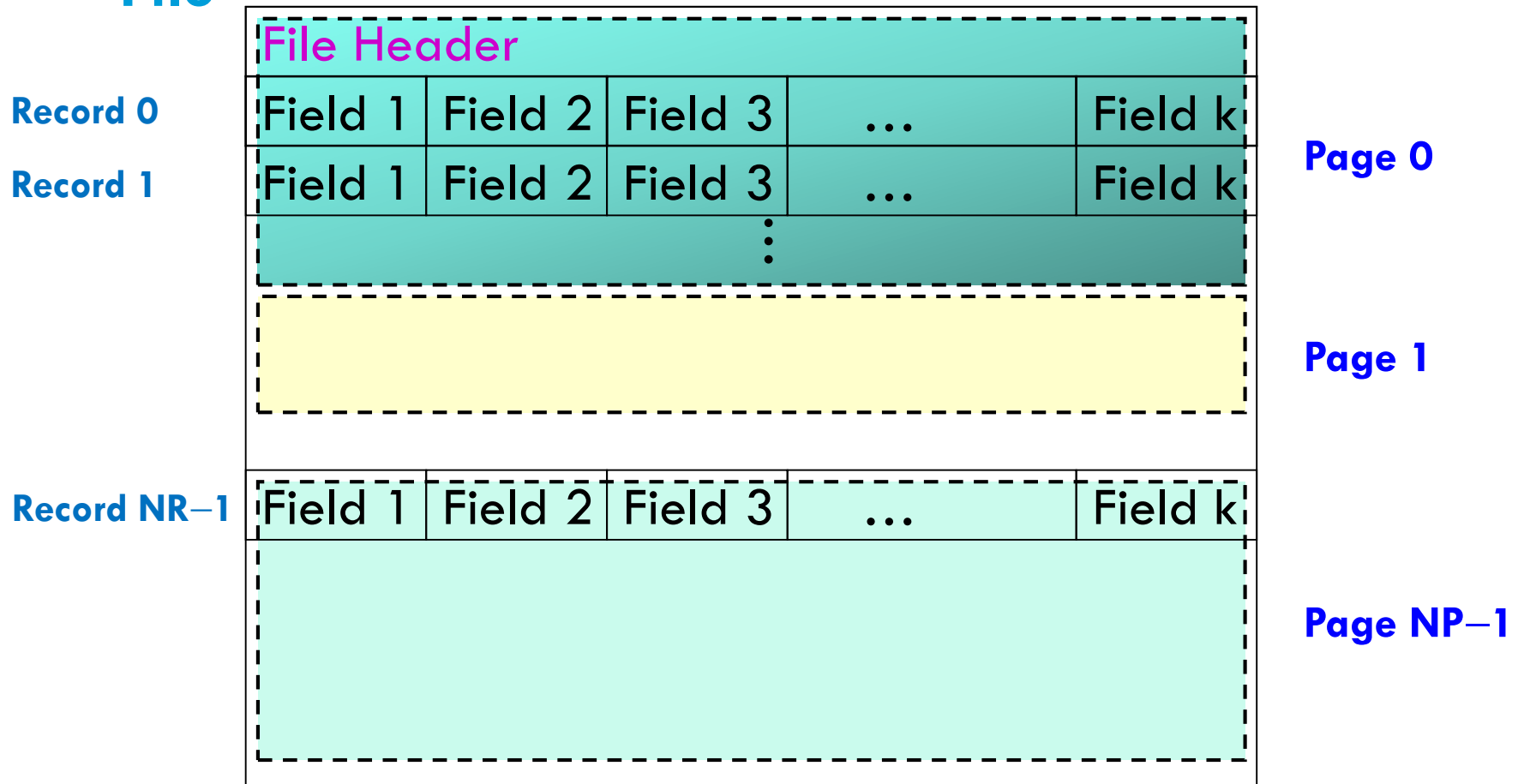
# Perché non usare sempre il file system?

- Le prestazioni di un DBMS dipendono fortemente dall'organizzazione fisica dei dati sui dispositivi di memorizzazione.
- Intuitivamente, l'allocazione dei dati dovrebbe mirare a ridurre i tempi di accesso ai dati e, a tale scopo è necessario conoscere come i dati dovranno essere elaborati (**logicamente**) e quali sono le **correlazioni logiche** tra i dati.
- Queste informazioni non possono essere note al file system.
  - Esempio:
    - se due relazioni, memorizzate su disco, contengono dati tra loro correlati mediante join può essere una buona idea memorizzarle in cilindri vicini, in modo da ridurre i tempi di seek;
    - se una relazione contiene attributi BLOB, può essere conveniente memorizzarli separatamente dagli altri attributi.

# Organizzazione dei dati nei file

**File**

## Schema di riferimento (semplificato)



# Rappresentazione dei valori

- Per ogni tipo di dati di SQL è definito un formato di rappresentazione interna, specifico nel contesto di un DBMS. Appresso alcuni possibili formati.
- **Stringhe a lunghezza fissa:** **CHAR(n)** (ipotesi di codifica ASCII)
  - si allocano n byte, eventualmente usando un carattere speciale per valori lunghi meno di n.

Esempio: se A è CHAR(5), 'cat' è memorizzato come cat␣␣.

- **Stringhe a lunghezza variabile:** **VARCHAR(n)** (ipotesi di codifica ASCII)
  - si allocano m+p byte, con m ( $\leq n$ ) byte usati per gli m caratteri effettivamente presenti e p byte per memorizzare il valore di m (per  $n \leq 255$   $p = 1$ ).

Esempio: se A è VARCHAR(10), 'cat' viene memorizzato in 4 byte come 3cat.

- **DATE** e **TIME** sono rappresentati **esternamente** con stringhe di lunghezza fissa (es. DATE: 10 caratteri YYYY-MM-DD; TIME: 8 caratteri HH:MM:SS) e **internamente** come sequenze di packed decimal digit (es. DATE: 4 byte, TIME: 3 byte) o altri formati numerici.
- **Tipi enumerati:** si usa una codifica intera

Esempio: week = {SUN, MON, TUE, ..., SAT} richiede un byte per valore:

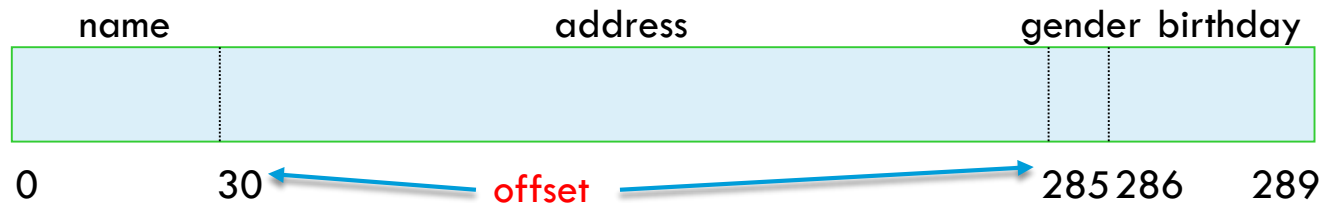
SUN: 00000001, MON: 00000010, TUE: 00000011, ...

# Record a lunghezza fissa

- Per ogni tipo di record nel DB deve essere definito uno schema (fisico) che permetta di interpretare correttamente il significato dei byte che costituiscono il record.
- La situazione più semplice si ha evidentemente quando tutti i record hanno lunghezza fissa, in quanto, oltre alle informazioni logiche, è sufficiente specificare l'ordine in cui gli attributi sono memorizzati nel record.

```
CREATE TABLE MOVIESTAR (  
  name      CHAR(30) PRIMARY KEY,  
  address   CHAR(255),  
  gender    CHAR(1),  
  birthdate DATE )
```

Ipotesi: stringhe con carattere speciale terminatore  $\perp$ .



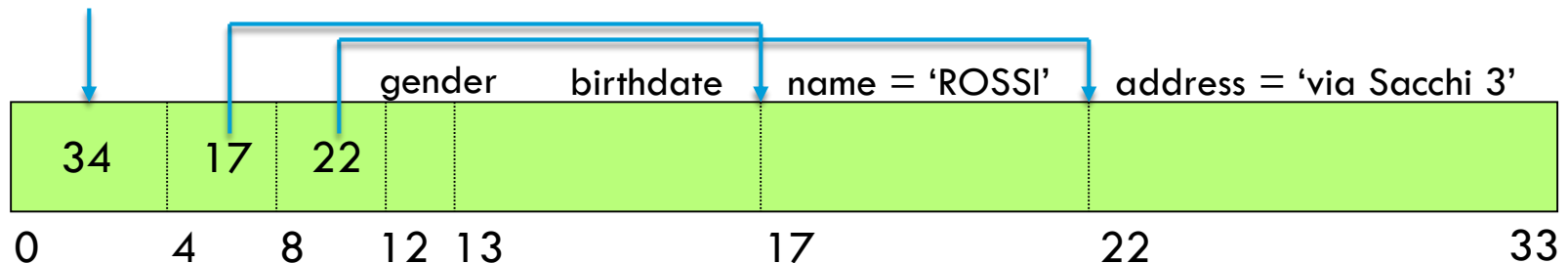
# Record a lunghezza variabile

- Il ricorso a record con lunghezza variabile è necessario in diverse situazioni, ad esempio: un file che contiene **record di tipo diverso**, o **record con attributi la cui lunghezza può variare**, o ancora **record con attributi multivalore**.
- Si hanno diverse alternative, che devono considerare anche i problemi legati agli aggiornamenti che modificano la lunghezza dei campi (e quindi dei record).
- Una possibile soluzione consiste nel memorizzare prima tutti i campi a lunghezza fissa, e quindi tutti quelli a lunghezza variabile; per ogni campo a lunghezza variabile si ha un “**prefix pointer**” che riporta l’indirizzo del primo byte del campo.

```
CREATE TABLE MOVIESTAR (  
  name    VARCHAR(30) PRIMARY KEY,  
  address VARCHAR(255),  
  gender  CHAR(1),  
  birthdate DATE )
```

La lunghezza dei dati è pari a 22 byte, ma nel suo complesso il record occupa 34 byte.

record length



# Record Header

- In generale ogni record include un **header** che, oltre alla lunghezza del record e ai riferimenti necessari per reperire campi a lunghezza variabile, può contenere altre informazioni tra cui:
  - ▣ l'identificatore della relazione cui il record appartiene;
  - ▣ l'identificatore univoco del record nel DB;
  - ▣ un **timestamp** che indica quando il record è stato inserito o modificato l'ultima volta;
  - ▣ stato del record (ad esempio se logicamente cancellato).
- Il formato di un header, ovviamente, è specifico del particolare DBMS.

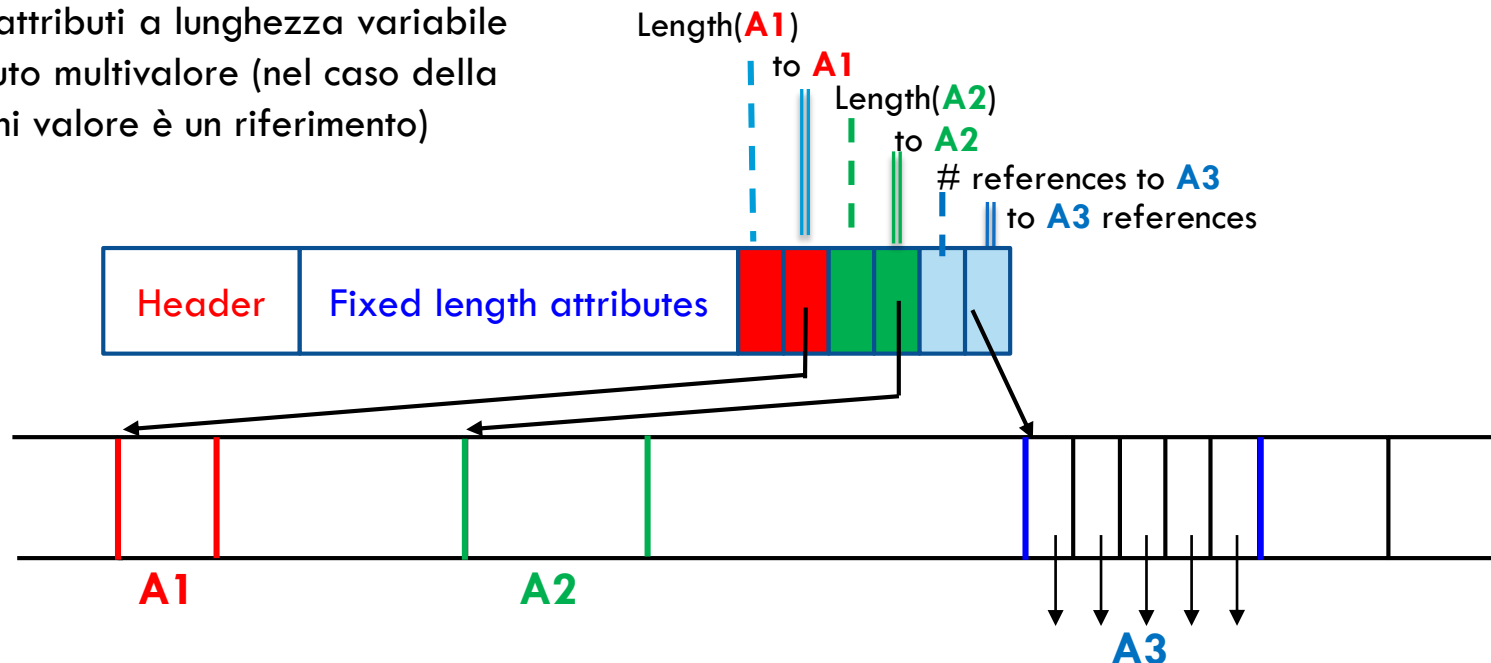
# Record con attributi multipli

- Merita particolare attenzione il problema della **gestione di record con attributi multivalore** (ciascuno eventualmente di lunghezza variabile).
- Una possibile soluzione che contempla entrambi i casi di attributi a lunghezza variabile e attributi ripetuti consiste nel memorizzare in un blocco separato le componenti a lunghezza variabile.

## Esempio:

**A1** e **A2** attributi a lunghezza variabile

**A3** attributo multivalore (nel caso della figura ogni valore è un riferimento)





# Organizzazione dei record nelle pagine

- Normalmente la dimensione di un record è (molto) minore di quella di una pagina.
- Nel caso di record a lunghezza fissa l'organizzazione in una pagina si potrebbe presentare come in figura:



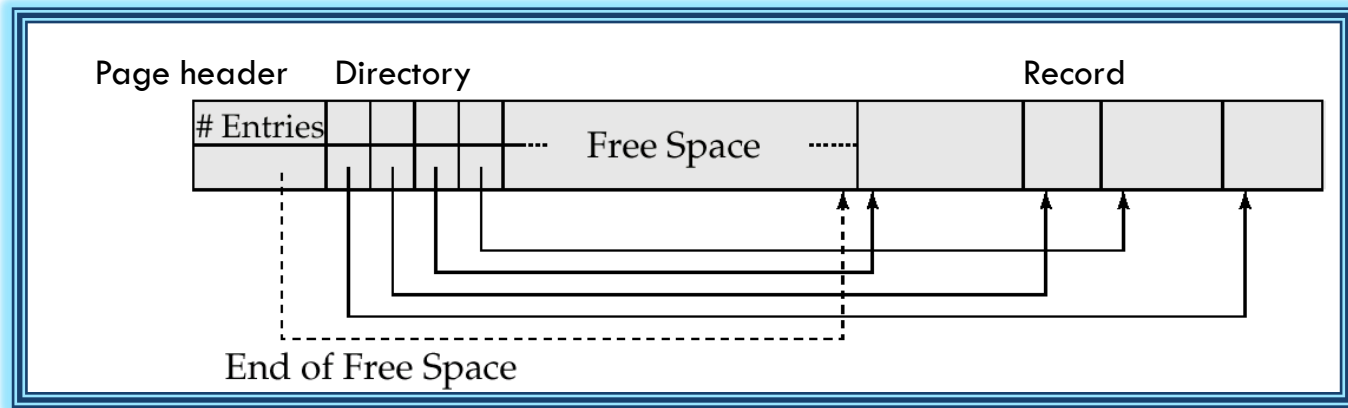
- Il page header mantiene alcune informazioni quali ad esempio:
  - ▣ ID della pagina nel DB, timestamp che indica quando la pagina è stata modificata l'ultima volta, relazione a cui le tuple nella pagina appartengono, ecc.
- Molto spesso un record è contenuto interamente in una pagina, quindi si può avere uno spreco di spazio.

# Un semplice esempio

- Nel caso visto prima, con record di lunghezza fissa pari a 290 byte, si supponga di usare pagine di dimensione  $P = 4 \text{ KB} = 4096 \text{ byte}$ .
- Supponendo che lo spazio header della pagina occupi 24 byte ne restano 4072 per i dati.
- Pertanto è possibile memorizzare in una pagina al massimo 14 record; infatti  $14 = \lfloor 4072/290 \rfloor$ : in ogni pagina resteranno quindi sempre inutilizzati almeno 12 byte.
- Se la relazione MOVIESTAR contiene 10000 tuple serviranno quindi almeno 715 pagine per memorizzarla;  $715 = \lceil 10000/14 \rceil$ .
- Se la relazione è memorizzata su disco e se la lettura di una pagina richiede in media 10 ms, la lettura di tutte le tuple comporterà un'attesa di circa 7.15 secondi (nel caso peggiore di letture random).

# Organizzazione a slot delle pagine

- Formato tipico di una pagina in un DBMS.



- La **Directory** contiene un **puntatore per ogni record nella pagina**.
- Con questa soluzione l'identificatore di un record, detto **RID** (**Row Identifier** oppure **Record Identifier**) o **TID** (**Tuple Identifier**) nel DB è formato da una coppia:
  - **PID**: identificatore della pagina;
  - **Slot**: posizione all'interno della directory.
- È possibile sia individuare velocemente un record, sia permettere la sua riallocazione nella pagina senza modificare il RID.

# Letture e scrittura di pagine

- La lettura di una tupla richiede che la pagina corrispondente sia prima trasferita in memoria, in un'area gestita dal DBMS detta **buffer pool**.
- Ogni buffer nel pool può ospitare una copia di una pagina su disco.
- La gestione del buffer pool, che è fondamentale dal punto di vista prestazionale, è demandata a un modulo del DBMS, detto **Buffer Manager (BM)**.
- Il **BM** è chiamato in causa anche nel caso di scritture, ovvero quando è necessario riscrivere su disco una pagina modificata.
- Il **BM** ha un ruolo fondamentale nella gestione delle transazioni, per garantire l'integrità del DB a fronte di guasti.

# Il Buffer Manager

- A fronte di una richiesta di una pagina, il Buffer Manager opera come segue (ipotesi: dimensione di un buffer = dimensione pagina):
  - Se la pagina è già in un buffer, si restituisce al programma chiamante l'indirizzo del buffer.
  - Se la pagina non è in memoria:
    - Il BM seleziona un buffer per la pagina richiesta. Se tale buffer è già occupato da un'altra pagina (rimpiazzamento), questa viene riscritta su disco solo se è stata modificata e non ancora salvata su disco e se nessuno la sta usando.
    - A questo punto il BM può leggere la pagina e copiarla nel buffer prescelto, rimpiazzando così quella prima presente.

# Interfaccia del Buffer Manager

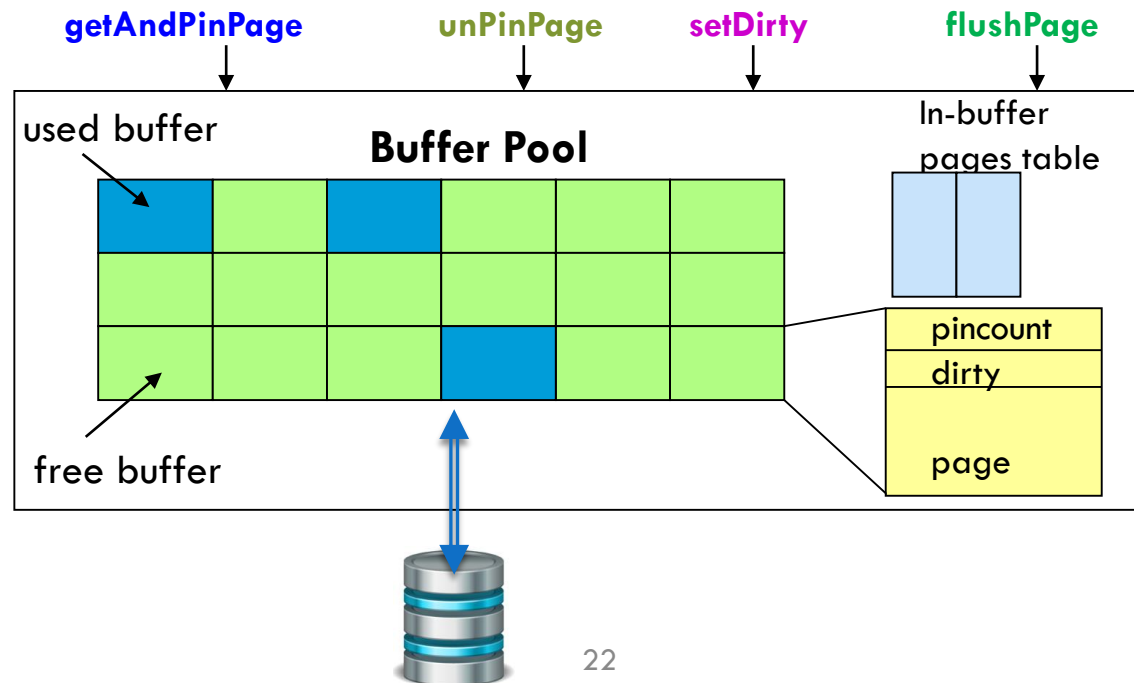
- L'interfaccia che il **BM** offre agli altri moduli del DBMS ha quattro metodi di base, che in modo semplificato possono essere così definiti:

**getAndPinPage:** richiede la pagina al **BM** e vi pone un **pin** (“spillo”), a indicarne l'uso;

**unPinPage:** rilascia la pagina e elimina un pin;

**setDirty:** indica che la pagina è stata modificata, ovvero è **dirty** (“sporca”);

**flushPage:** forza la scrittura della pagina su disco, rendendola così **clean** (“pulita”).



# Politiche di rimpiazzamento

- Nei sistemi operativi una comune politica adottata per decidere quale pagina rimpiazzare è **LRU (Least Recently Used)**, ovvero si rimpiazza la pagina che da più tempo non è in uso.
- **Nei DBMS LRU non è sempre una buona scelta**, in quanto per alcune query il “*pattern di accesso*” ai dati è noto, e può quindi essere utilizzato per operare scelte più accurate, in grado di migliorare anche molto le prestazioni.
- Il valore **hit ratio**, ovvero la frazione di richieste che non causano un’operazione di I/O su disco, indica sinteticamente quanto buona è una politica di rimpiazzamento.
  - ▣ **Esempio**: esistono algoritmi di join che scandiscono N volte le tuple di una relazione. In questo caso la politica migliore sarebbe **MRU (Most Recently Used)**, ovvero rimpiazzare la pagina usata più di recente.

# Domande?

---

