Tecnologie delle Basi di Dati M

Antonio Davide Calì

6 marzo 2014

WWW.ANTONIOCALI.COM

Anno Accademico 2013/2014

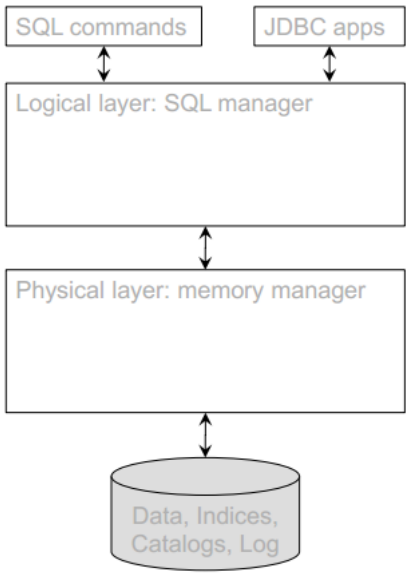
Professori: Marco Patella, Paolo Ciaccia

Indice

Parte I
Struttura Fisica Database

1 Architettura di un Database

Figura 1:



Come si vede dalla figura (Fig. 1) un database è formato da più livelli.

Dati Nella sezione dati sono presenti i contenuti del database (appunto i dati), ma inoltre anche gli indici, i cataloghi e i log. I **cataloghi** sono tabelle contenenti *metadati*, cioè si intendono tutte le informazioni utili per il database quali nomi tabelle, nomi attributi di una singola tabella, numero di tuple in una tabella etc... I **LOG** invece altro non sono che un registro di tutte le attività svoltesi all'interno del database (anche questi ultimi vengono gestiti tramite tabelle

Livello Fisico Il livello fisico di occupa di gestire i dati, ad esempio memoria di massa (memoria secondaria/terziaria) e memoria principale

Livello Logico Gestisce *SQL*, trasformando sql stess o in istruzioni per accedere al *livello fisico*

Stato applicativo Viene usato questo stato per permettere all'utente di accedere al *livello logico*

Aver suddiviso il database in più livelli permette un livello di astrazione maggiore, con questo si intende la possibilità di modificare un determinato livello senza dover apportare modifiche ad un altro poichè quest'ultimo non è conscio di come sia realizzato il livello sottostante, ma ne utilizza solo determinate caratteristiche che dall'esterno rimangono invariate. Oltre ai livelli prima descritti, in questo approccio vengono introdotte anche le *viste.*

Le **viste** descrivono il punto di vista degli utenti, esse permettono di scegliere quali dati mostrare ad un certo tipo di utenti, dunque che "vista" del database essi avranno, infatti non è opportuno che un normale utente abbia accesso a tutti i dati del nostro database, dunque è possibile limitare queste informazioni; inoltre le viste permettono un'astrazione rispetto al livello logico, poichè potrebbe essere necessario modificare lo schema logico (per aggiungere magari una tabella o un attributo ad una già esistente), senza però andar a inficiare il comportamento e la visualizzazione dei dati permessi all'utente. Ad esempio se in precedenza era stata sviluppata un'applicazione che mostrasse solo un determinato set di dati, modificando il livello logico, l'applicazione non risente dell'accaduto poichè avrà ancora a disposizione i dati a lei disponibili. L'astrazione tra le *viste* e il *livello logico* viene detta **indipendeza logica.**

Lo **schema logico** *definisce la struttura logica del database.* In questo schema dunque vengono realizzate le relazioni con gli attributi ad esse associate, i vincoli presenti sui singoli attributi e i vincoli invece che devono sussistere tra una relazione e un'altra. Anche in questo caso si presenta un'*astrazione* rispetto allo schema fisico, infatti, potrebbe essere necessario modificare il livello fisico senza però che il livello logico risenta di questa modifica. Ad esempio se volessi introdurre un *indice* per una determinata relazione per migliorarne la performance è necessario modificare lo schema fisico, la modifica non viene risentita dal livello logico, il quale continua a mantenere le informazioni che compongono il database inalterate. L'astrazione tra *livello logico* e *livello fisico* viene detta **indipendenza fisica.**

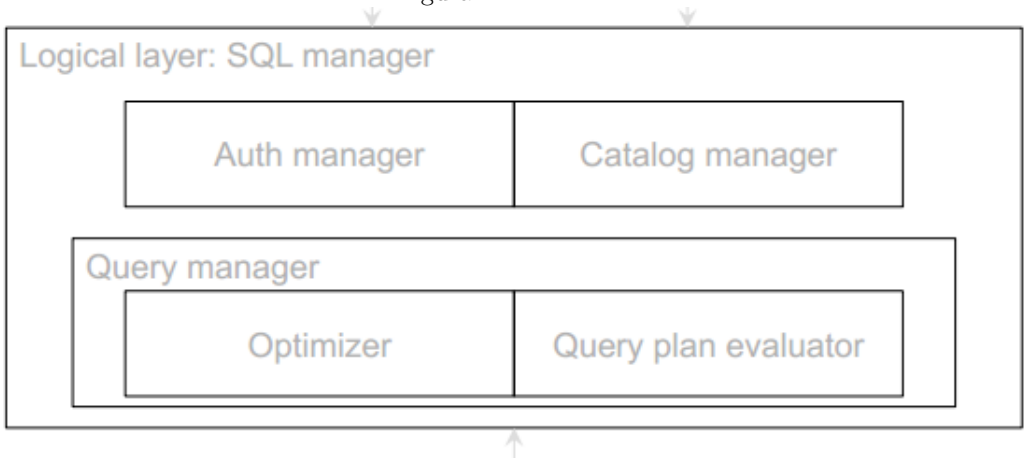
Lo **schema fisico** *descrive come i dati sono effettivamente memorizzati sul disco.*

1.1 Livello Logico

Il livello logico (Fig. 2) si suddivide in

- **Auth Manager:** Gestore degli accessi. Molto simile al gestore degli accessi presente in un qualsiasi sistema operativo.
- **Catalog Manager:** Gestore dei cataloghi. I cataloghi vengono gestiti come tabelle, dunque possono essere gestiti a livello logico. Inoltre risulta essere anche il primo gestore ad essere interpellato in quanto viene chiamato per controllare la semantica e la sintassi delle istruzioni SQL, ad esempio se faccio una query in una tabella X, devo controllare che effettivamente esista una tabella X.

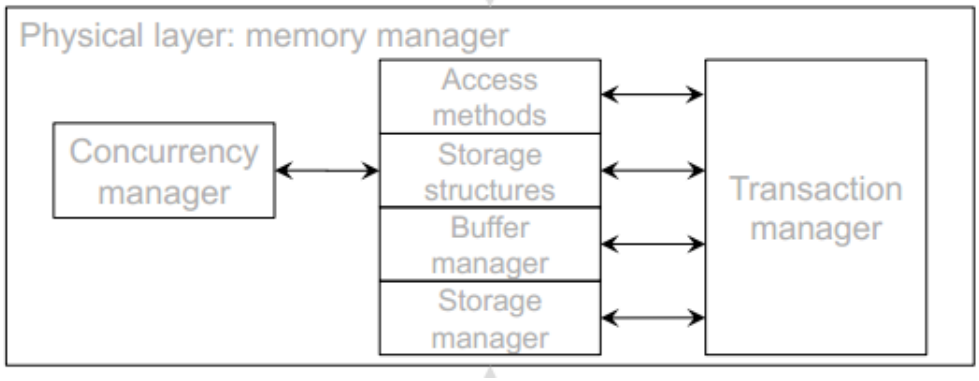
Figura 2:



- **Query Manager**: Gestore delle query. Il gestore delle Query si suddivide ulteriormente con *Optimizer* e *Valutatore dei piani di accesso.* Questi svolgono un ruolo importante nella risoluzione delle query, in quanto l'optimezer sceglie fra i più modi possibili quello che ha performance migliore, ma per far questo si basa su statistiche che il Valutatore di piani fornisce.

1.2 Livello Fisico

Figura 3:



Il livello fisico (Fig. 3) si suddivide in 3 blocchi

- **Gestore della concorrenza:** come accade anche in un *sistema operativo* è necessario gestire la concorrenza per l'accesso alla memoria.

- **Gestore delle transazioni:** serve a gestire le transazioni. Una *transizione* è una lista di azioni che non possono essere viste come indipendenti una dall'altra.

Questi due gestori si interfacciano poi a una terza sezione anch'essa suddivisa in

- **Gestore della memoria:** si occuppa di gestire effettivamente la memoria (secondaria).
- **Gestore del buffer:** gestisce in maniera propria i *buffer*, in quanto non è opportuno lasciare la gestione della memoria al sistema operativo a causa dell'inefficienza. Ciò che succede all'interno della memoria principale è che solitamente sia presente una sola applicazione in attività, il *dbms*, e poichè il dbms utilizzza il gestore del buffer, andrà ad ottimizzare la gestione dello stesso.
- **Strutture di memorizzazione:** serve a mettere in comunicazione i "dati" (scritti in bit) con le query SQL, cioè serve a dare struttre logiche ai dati per poi essere memorizzati sulla base fisica.
- **Metodi di accesso:** viene utilizzato per accedere nei migliore dei modi (poichè esistono diversi modi) ai dati presenti a livello fisico.

1.3 Utenti di un DBMS

Gli utenti che accedono a un DBMS si possono suddividere in diverse categhorie

1. Utenti delle applicazioni: non hanno alcuna conoscenza di come sia realizzato un DBMS e/o il database a cui si interfacciano, utilizzano solo l'applicazione fornita loro per accedere i dati presenti sul database stesso (e.g. segretarie)
2. Utenti non programmatori: utenti che hanno conoscenza di come sia realizzato il *database* sul quale lavorano, hanno un approccio interettivo con il *database* utilizzando un *Data Manipulation Language* (DML), una famiglia di linguaggi che consente di leggere, inserire, modificare o eliminare i dati in un database. (e.g. SQL)
3. Programmatori di applicazioni: hanno conoscenza di come sia realizzato il *database* tale da poter realizzare un'applicazione per permettere la fruibilità a utenti tipo 1. Utilizzano API quali *JDBC* per connettersi al database.
4. Progettisti di database: hanno le conoscenze tali da poter realizzare lo schema concettuale/logico (e dunque gli schemi ER) per un determinato micromondo. Il loro compito termina con la realizzazione dello schema logico utilizzando un *Data Definition Language* (DDL), un linguaggio che permette di creare, modificare o eliminare gli oggetti in un database ovvero agire sullo schema logico di database.

5. Amministratori di database: hanno forti conoscenze sia sul *database* per il quale sono amministratori sia per il *dbms* utilizzato per la loro realizzazione. Sarà loro compito andare a modificare anche lo *schema fisico* per ottenere un buon tuning del database.

6. Progammatori di DBMS: ottime conoscenze nel settore, realizzano dbms.

1.4 Memoria e la sua gestione

Come risaputo La memoria di un calcolatore è organizzata in una gerarchia a 3 livelli:

1. Memoria principale (RAM)

2. Memoria secondaria (dischi magnetici)

3. Memoria terziaria (nastri e jukebox)

Sarebbero presenti anche una *memoria interna* (cache e registri) e una *memoria offline*, ma di poca importanza per la realizzazione di un *dbms*.

La **memoria principale** è una memoria *molto veloce* (tempo accesso: ~50ns - velocità: ~3 GB/s), **volatile**, ma di *spazio limitato* (Capacità: ~1 GB) e di un *costo monetario non indifferente* (Costo: ~30 /GB); è la memoria dove risiede il programma in esecuzione.

La **memoria secondaria** è caratterizzata da una *velocità ridotta* (Tempo di accesso: ~5 ms - Velocità: ~120 MB/s), **permanente** (conserva i dati), ma di *capacità* superiore (Capacità: <2 TB) e *costo monetario contenuto* (Costo: ~0.10 /GB); è la memoria dove solitamente risiedono i dati.

La **memoria terziaria** è molto simile alla memoria secondaria ma con velocità minori (Tempo di accesso: ~30s - Velocità: ~3 MB/s); viene impiegata solitamente solo per effettuare dei backup.

Aver questa differenziazione di memoria implica una suddivisione anche del DBMS/DB. Infatti i dati del database risidieno in *memoria secondaria* a causa della grandezza del database stesso, però, per l'elaborazione dei dati stessi il *dbms* (in esecuzione sulla memoria principale) ha bisogno di portare i dati da *memoria secondaria* a *memoria principale* per poterli elaborare. Questo trasporto genera rallentamenti impossibili da evitare ma che se ben strutturati possono portare ugualmente a una buona performance.

Un **blocco** (o anche detto **pagina**) è l'unità di dati che vengono trasportati, quindi, non si trasportano singole tuple, ma bensì un insieme di *tuple* (appunto i blocchi).
Per riuscire a mantenere una buona performance si rende necessaria una buona implementazione del *database* attraverso:

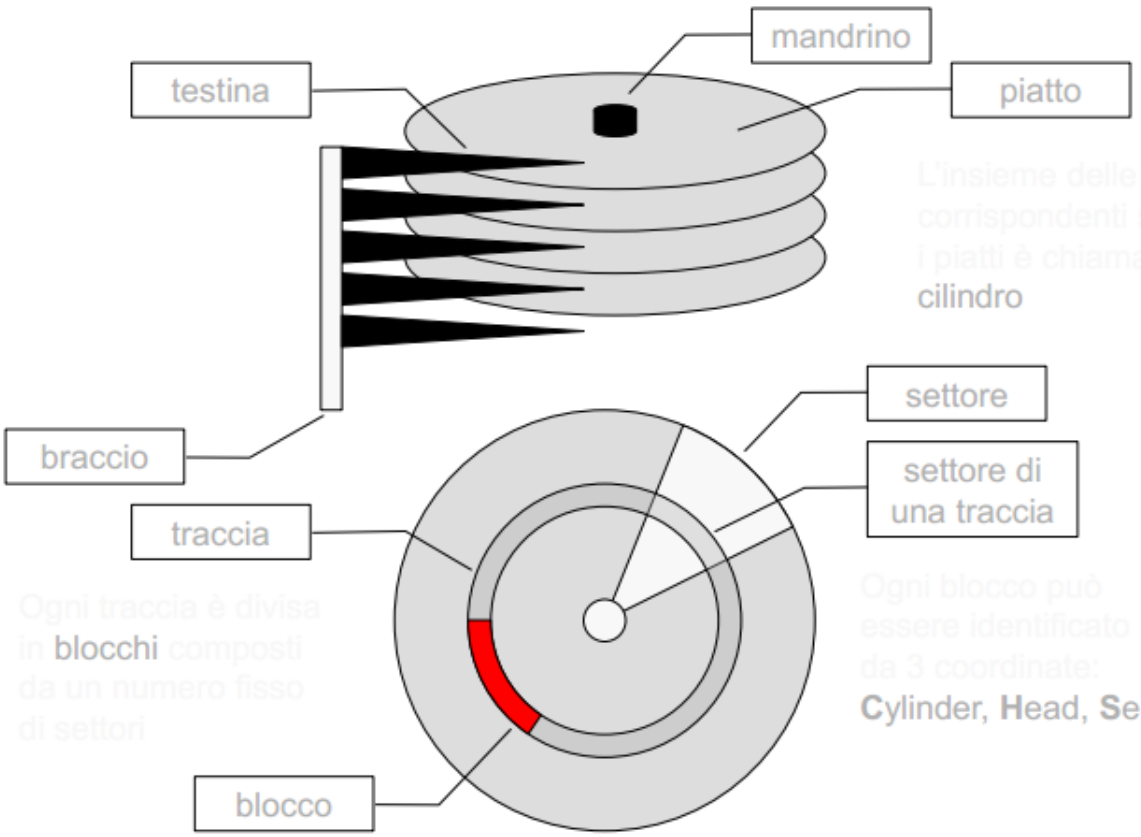
- Organizzazione efficiente delle tuple su disco

- Strutture di accesso efficienti

- Gestione efficiente dei buffer in memoria
- Strategie di esecuzione efficienti per le query

1.4.1 Struttura di un Hard Disk

Figura 4:



Testina Su ogni piatto è presente una testina per accedere in scrittura o in lettura ai dati memorizzati sul piatto; la posizione di tale testina è solidale con tutte le altre sugli altri piatti. In altre parole, se una testina è posizionata sopra una traccia, tutte le testine saranno posizionate nel cilindro a cui la traccia appartiene

Piatto un disco rigido si compone di uno o più dischi paralleli, di cui ogni superficie, detta "piatto" e identificata da un numero univoco, è destinata alla memorizzazione dei dati.

Traccia ogni piatto si compone di numerosi anelli concentrici numerati, detti tracce, ciascuna identificata da un numero univoco.

Cilindro l'insieme di tracce alla stessa distanza dal centro presenti su tutti i dischi o piatti è detto cilindro. Corrisponde a tutte le tracce aventi il medesimo numero, ma diverso piatto.

Settore ogni piatto è suddiviso in settori circolari, ovvero in "spicchi" radiali uguali ciascuno identificato da un numero univoco.

Blocco l'insieme di settori posti nella stessa posizione in tutti i piatti.

Un **blocco** viene dunque identificato dalla tripla **C**ylinder, **H**ead, **S**ector (CHS), dunque per accedere a determinati dati basta indicare il cilindro, la testina e il settore a cui appartengono i dati stessi.

1.4.2 Prestazioni di HardDisk

Le prestazioni di un HD sono sia interne che esterne ed esse dipendono da:

- Prestazioni interne:
  - HardDisk in se
  - Caratteristiche meccaniche (e.g. velocità di movimento della testina, rotazione dei dischi)
  - Tecniche di memorizzazione e codifica dei dati
  - Disk controller (cioè l'interfaccia tra la parte hardware del disco e il sistema di calcolo) e le prestazioni dipendono da:
    - Accetta comandi di alto livello per leggere/scrivere settori e per controllare il meccanismo
    - Aggiunge informazioni di controllo degli errori (e.g. checksum) nei settori
    - Verifica la correttezza di ciò che ha appena scritta rileggendo i settori
    - Esegue il mapping tra indirizzi logici dei blocchi e settori sul disco
- Prestazioni esterne:
  - Tipo di interfacca
  - Architettura del sistema I/O
  - File System utilizzato

Prestazioni interne Ciò che incide di più sulla prestazione interna è la *latenza* cioè il tempo necessario per raggiungere le informazioni di interesse. Il tempo di latenza è composto da:

1. **Command Overhead Time**: cioè il tempo necessario per impartire un ordine (e.g. leggi/scrivi) al drive, tipicamente di *0.5ms* e può essere trascurato

2. **Seek Time (TS):** tempo impiegato dalla testina per posizionarsi dalla traccia attuale alla traccia desiderata. Il tempo medio è di *2-10ms.* Dato il tempo massimo di seek (cioè il tempo che la testina impiega per spostarsi dalla traccia più interno a quella più esterna) il tempo medio di seek è di circa il tempo massimo di seek. Inoltre il tempo di seek per la scrittura è di circa *1ms* superiore al tempo di lettura, questo perchè in scrittura bisogna essere perfettamente allineati con la traccia.

3. **Settle Time:** tempo di stabilizzazione del braccio (a causa delle oscillazioni)

4. **Rotational Latency (TR):** tempo necessario per raggiungere il blocco di interesse. Il caso migliore è se la testina si trova proprio sul blocco che si vuole leggere, il caso migliore invece se il blocco desiderato è appena passato e dunque ci vuole un intero giro del disco per riportare il blocco sotto la testina. La latenza rotazionale media è di ¡ rispetto al caso peggiore. Il tempo varia da *2 a 11ms.* La formula per calcolare la latenza rotazionale è $\frac{60}{Velocità\,di\,rotazione} \cdot 0,5 \cdot 1000$

Transfer Rate Il *transfer rate* è la velocità massima alla quale il drive può leggere o scrivere i dati. Tipicamente è dell'ordine di qualche decina di MB/s e si riferisce alla velocità con cui si trasferiscono bit dai (sui) piatti dulla (dalla) cache del controller. Si può stimare come

$$\frac{\left(\frac{bytes}{settore}\right) \cdot \left(\frac{settori}{traccia}\right)}{tempo\,di\,rotazione}$$

Esempio: ho 512 bytes per ogni settore e 368 settori per ogni traccia, la velocità di rotazione è di 7200 rpm (round per minute), il transfer rate dunque è uguale a $\frac{(512*368)}{(60/7200)} = 21,56MB/s$. Come si nota la velocità di trasferimento è in realtà minore rispetto alla velocità nominale.

Pagina Un **blocco** (o **pagina**) è una *sequenza contingua di settori su una traccia* (essendo settori contigui possono essere dunque letti/scritti in maniera consecutiva senza bisogno di spostare la testina) e costituisce l'unità di I/O per il trasferimento di date da/per la memoria principale. Siccome il tempo per la lettura/scrittura di dati su disco è dominato dal movimento meccanico della testina, scrivendo/leggendo settori contigui riusciamo a migliorare la performance,

ecco perchè la pagina rappresenta l'unità di trasferimento. La dimensione tipica di una *pagina* è di pochi KB (circa 4-64KB). Bisogna notare che *pagine piccole* comportano un maggior numero di operazioni di I/O (servono più operazioni per leggere una stessa informazione distribuita però su più pagine), mentre *pagine grandi* portano ad avere una frammentazione interna (poichè non tutte le pagine vengono riempite) e richiedono più spazio in memoria centrale per essere caricate.
Il *tempo di trasferimento di una pagina* (**TT**) da disco a memoria centrale dipende dalla *dimensione della pagina* (**P**) e dal *tranfer rate* (**TR**). $TT = \frac{P}{TR}$
Esempio: TR=21.56MB/s - P=4KB - $TT = \frac{4}{(21.56*1024)} = 0.18ms$

2 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 (es: 4 KB). I file hanno una dimensione multipla della dimensione della pagina. Ogni pagina memorizza più *record* (corrispondenti alle tuple logiche) i quali a loro volta consistono di più *campi* di lunghezza fissa e/o variabile che rappresentano gli attributi delle nostre *relazioni.*

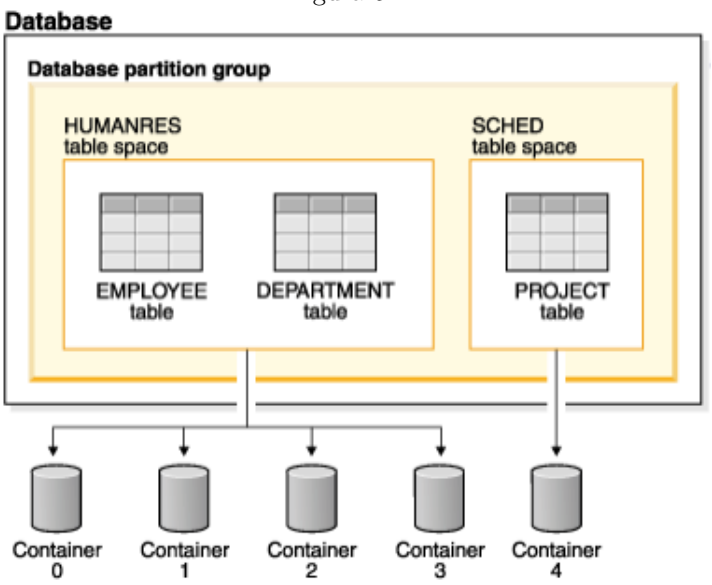
I "file" del *dbms* non necessariamente corrispondono ai file del file system del sistema operativo, infatti potrebbe (molto spesso lo è) essere inopportuno che il *dbms* affidi al sistema operativo la gestione dei file (così come è inopportuno che il *dbms* affidi al sistema operativo la gestione della memoria principale); di conseguenza il *dbms* potrebbe avere un file system a sé dedicato (più articolato ma più flessibile) per la gestione interna dei suoi file.
Casi limite:

- ogni *relazione* del database è memorizzata in un proprio file
- tutto il database è memorizzato in un singolo file

2.1 Modello DB2

DB2 organizza lo spazio fisico in *tablespace*, ognuno dei quale è una collezione di *container*. L'esempio in figura (Fig 5) mostra un database con 3 relazioni (tabelle): *EMPLOYEE, DEPARTMENT e PROJECT.* Come si vede employee e department appartengono ad uno stesso *tablespace* HUMANRES, mentre project appartiene al *tablespace* SCHED. All'atto di progettazione infatti ci si è accorti che le tabelle di employee e department vengono spesso usate assieme, di conseguenza, per motivi di performance è stato deciso di assegnarle ad uno stesso *tablespace,* mentre project che verrà usato da solo o raramente associato alle altre due tabelle è stato inserito in un *tablespace* dedicato. Si ha un vantaggio in flessibilità infatti è possibile memorizzare cose diverse in dispositivi diversi (*cointainer*) e inoltre l'aggiunta di nuove tabelle può avvenire aggiungendo *tablespace.* Si noti che ogni **cointainer** può essere un *dispositivo*, una *directory* o un *file* (in caso un container sia una directory o un file ci si appoggia al file

Figura 5:



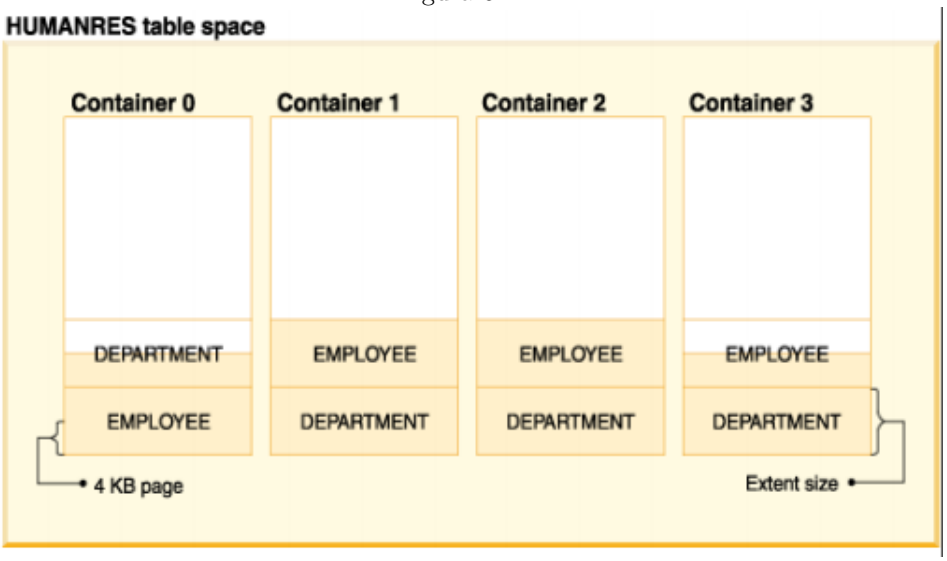
system associato). Si noti che ogni *relazione* è memorizzata in un singolo *tablespace,* ma un *tablespace* può contenere più relazioni, così come per i *container*, cioè ogni *container* sta in un solo *tablespace*, ma un *tablespace* può contenere più *container*. Supponendo che i 4 *container* associati a HUMANRES siano 4 dischi/dispositivi diversi, avremmo un'ottima efficienza e flessibilità, in quanto è possibile memorizzare tutte le tuple di Employee sul container 0 e tutte le tuple di Departmente sul container 1, o ancora suddividere invece le tuple di employee e deparment sui vari *container.* L'obiettivo è di ottenere la miglior performance possibile nella risoluzione delle query.
Il *dbms* effettua inoltre in maniera automatica il bilanciamento dei dati nei *cointainer* in modo da avere una percentuale di occupazione pressochè uguale (anche a fronte di cointainer di natura diversa).

Ogni cointeiner è diviso in **extent**, che rappresentano l'unità minima di allocazione su disco e sono costituiti da *insiemi cointigui di pagine* di 4KB (valore di default di **P**). La dimensione di un extent dipende dallo specifio tablespace e viene scelta all'atto della creazione di quest'ultimo. **Ogni extent contiene dati di una sola relazione**.

Ogni database deve avere almeno tre *tablespace* che vengono utilizzati per memorzzare dati diversi:

- cataloghi (*system tablespace*, vengono usati per i cataloghi)
- tabelle utente (*uno o più user tablespace*, vengono usati per i dati)

Figura 6:



- tabelle temporanee (*uno o più user tablespace*, utilizzati per memorizzare dati temporanei ad esempio perchè la ram non riesce a mentenere tutta una tabella in memoria ma necessaria alla completazione di join)

2.1.1 Tipi di Tablespace

Esistono tre tipi differenti di tablespace:

1. **SMS** (System Manged Space): la gestione dello storage è assegnata al sistema operativo
2. **DMS** (Database Managed Space): la gestione dello storage è affidata all'utente
3. **AS** (Automatic Storage): la gestione dello storage è demandata al *dbms*

SMS

Creazione: CREATE TABLESPACE ... MANAGED BY SYSTEM

Definizione dei container: indicando il nome di una directory, il sistema operativo userà la directory indicata (potrebbe anche essere la root)

Allocazione iniziale: data in gestione al sistema operativo, probabile frammentazione

Modifica ai cointainer: non permessa, in quanto tutto è gestito dal sistema operativo

Richiesta di memoria aggiuntiva: possibile fino all'esaurimento del file system

Manutenzione richiesta: nessuna, tutto è gestito dal sistema operativo (vantaggio perchè non devo gestire nulla, svantaggio perchè mi rende impossibile anche la manutenzione in caso di necessità)

Dimensione massima: n*dimensione massima di un file

Separabilità di oggetti: non permessa, tabelle e inidici nello stesso tablespace

Vantaggi: Su tabelle piccole, c'è il controllo di dove vengano memorizzati i dati (la directory indicata) e c'è il controllo sulla situazione dello storage

DMS

Creazione: CREATE TABLESPACE ... MANAGED BY DATABASE

Definizione dei cointainer: indicando un dispositivo o un file, in caso di file bisogna stabilire la dimensione

Allocazione iniziale: all'atto di creazione del tablespace (se viene stabilito che un container sia un device allora è improbabile la frammentazione)

Modifica ai cointainer: è possibile (anzi è necessario) aggiungere e/o rimuovere container (in questo caso il *dbms* provvede automaticamente al ribilanciamento). **NB** l'aggiunta dei container è necessaria se ci accorgiamo che lo spazio nel tablespace sta per esaurirsi, la rimozione quando invece ci accorgiamo che i container sono troppo "vuoti"

Richiesta di memoria aggiuntiva: i cointainer possono essere estesi

Manutenzione richiesta: spetta all'amministratore capire quando e se aggiungere e/o rimuovere container. Inoltre in caso si vogliano mantenere il numero dei container inalterato ma ci si accorge che c'è uno sbilanciamento di occupazione di memoria fra essi è a carico dell'amministratore richiedere il ribilanciamento.

Dimensione massima: 512GB a causa dell'indirizzamento delle tuple in memoria. Possibile anche 64TB per le large tablespace, con occupazione di 7bit in più per l'indirizzamento delle tuple.

Separabilità di oggetti: gli oggetti (es. tabelle e indici) possono essere memorizzati in tablespace diversi. (Può portare a un forte aumento della performance dovuta a parallelizzazione delle operazioni, es. Indice mi indica la posizione della tupla e nel frattempo leggo anche la tupla stessa)

Vantaggi: Su tabelle grandi, c'è il controll di dove i dati vengano memorizzati, c'è il controllo sulla situazione dello storage, inoltre è possibile memorizzare oggetti diversi (es. indici, tabelle) in tablespace differenti

AS

Creazione: CREATE TABLESPACE ... [MANAGED BY AUTOMATIC STORAGE] (Il campo è opzionale visto che è l'opzione di default)

Definizione dei container: automatica, essendo dovuta al fatto che all'atto di installazione del *dbms* esso si riserva dello spazio in memoria, automaticamente il *dbms* stesso utilizza parte di questo spazio per la gestione dei container.

Allocazione iniziale: Se si tratta di *system tablespace* o *user tablespace* avviene all'atto della creazione del tablespace stesso. In caso di un *tablespace temporaneo* esso viene allocato solo quando necessario.

Modifica ai cointainer: gestita in maniera automatica dal *dbms*

Richiesta di memoria aggiuntiva: il *dbms* automaticamente estende i cointainer in caso di bisogno

Manutenzione richiesta: in caso si voglia ridurre un determinato *tablespace* l'amministratore deve intervenire. Vale inoltre lo stesso discorso del ribilanciamento descritto su **DMS**

Dimensione massima: corrisponde alla dimensione del file system utilizzato per la directory specificata, l'AS si prende tutto lo spazio disponibile nella directory

Separabilità di oggetti: gli oggetti (es. tabelle ed indici) possono essere memorizzati in tablespace diversi

Vantaggi: su tabelle grandi, viene semplificata la gestione della crescita dei container, è possibile la memorizzazione di oggetti diversi (es. tabelle, indici) in tablespace differenti.

2.1.2 Attributi del tablespace

All'atto di creazione di un tablespace è possibile specificare una serie di parametri, tra cui:

EXTENTSIZE: "grandezza" (numero di blocchi consecutivi) dell'*extent*

BUFFERPOOL: nome del pool di buffer associato al tablespace, permette di riservare un buffer di memoria a determinati dati che reputo più importanti piuttosto che mischiarli assieme a tutti gli altri

PREFETCHSIZE: numero di pagine da trasferire in memoria prima che vengano effettivamente richieste, cioè permette di specificare il trasferimento di più pagine rispetto magari a quelle davvero necessarie

OVERHEAD: stima del tempo medio di latenza per un'operazione I/O

TRANSFERRATE: stima del tempo medio per il trasferimento medio di una pagina

I parametri **OVERHEAD** e **TRANSFERRATE** sono parametri della memoria secondaria, che vengono forniti anche al tablespace, questo a favore della performance in quanto sono i parametri su cui si basa il *valutatore di piani* per creare la stima di quale sia il modo migliore per la risoluzione di una query (vengono dunque usati dall'ottimizzatore). Siccome in un tablespace possono esserci diversi container, e questi possono corrispondere a dischi, in caso di dischi di qualità diversa, quindi con overhead e transferrate diversi, bisogna impostare quelli del disco peggiore.

Ci si potrebbe chiedere perchè non usare sempre il file system del sistema operativo. Le prestazione di un *dbms* dipendono fortemente da come i dati sono organizzati sul disco. Intuitivamente l'allocazione dei dati dovrebbe mirare a ridurre i tempi di accesso ai dati e per far questo bisogna sapere come i dati dovranno essere elaborati e quali sono le relazione logiche tra i dati stessi. Tutte queste informazioni non possono essere note al file system.

Esempi:

- Se due relazioni 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 (Binary large object, sono tipi di dati binari molto grandi per la memorizzazione di dati come ad esempio la codifica in JPEG di una immagine) può essere una buona idea memorizzarli separatamente dagli altri attributi.

2.2 Organizzazione dei dati

Supponiamo per semplicità (vedi Fig. 7) di avere record delle stesse dimensioni. Come si vede è presente un *file header* che contiene informazioni sul file stesso e di seguito i record, uno dopo l'altro, i quali sono organizzati in *pagine* (Page) o in *extent* (gruppi contigui di pagine).

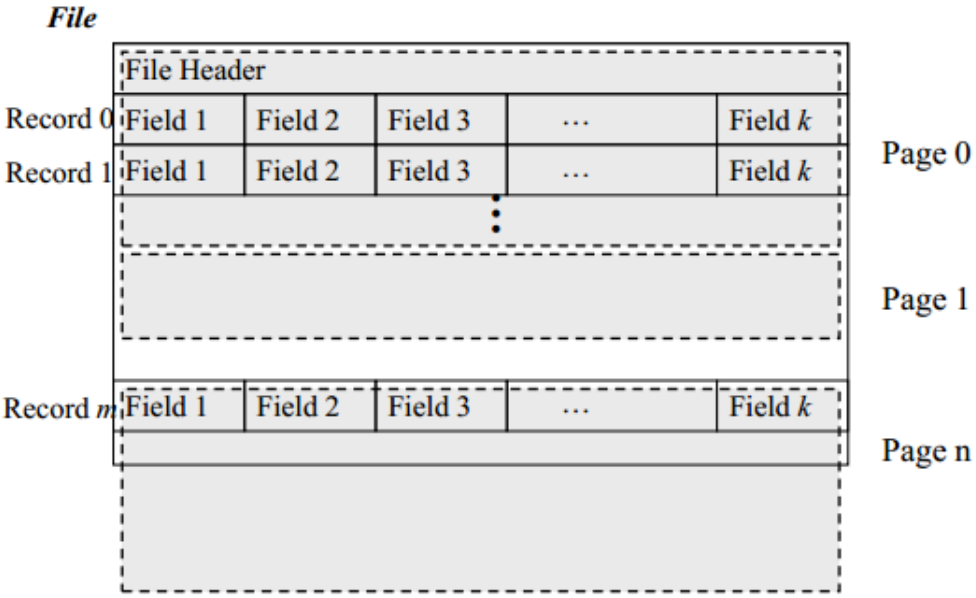
2.2.1 Rappresentazione dei valori

Ogni tipo di dato di SQL ha una rappresentazione propria.

- **INT**: tipica rappresentazione standard in cui dato un calcolatore fatto di una parola di n bit permette numeri fino a $2^n - 1$
- **DOUBLE**: rappresentazione in virgola mobile secondo lo standard IEEE 754[1]

[1] http://it.wikipedia.org/wiki/IEEE_754

Figura 7:



- **CHAR(n)**: sono le stringhe di esattamente n caratteri. Per la loro rappresentazione si usano n byte, eventualmente usando un carattere speciale *null* per valori più corti di n.
  Esempio: se A è CHAR(5), 'cat' viene memorizzato come cat¶¶ dove ¶ in questo caso rappresenta il *null*

- **VARCHAR(n)**: sono strinche di lunghezza di massimo n caratteri. Per la rappresentazione si allocano $m+p$ byte, dove p byte sono effettivamente i byte che rappresentano la stringa, m byte (che può essere anche un singolo byte in caso $p < 2^8 = 256$byte) che rappresenta la lunghezza della stringa. Viene memorizzato in testa la lunghezza della stringa e successivamente la stringa stessa.
  Esempio: se A è VARCHAR(10), 'cat' viene memorizzato in 4byte (1byte per dire la lunghezza [3], 3byte per effettivamente la parola 'cat') come '3cat'

- **DATE**: tipo di dato che identifica una data. Viene rappresentato con stringhe di lunghezza fissa di 10 caratteri siffatte *YYYY-MM-DD* (YYYY: anno, MM: mese, DD: giorno)

- **TIME**: tipo di dato che identifica un'ora. Viene rappresentato con stringhe di lunchezza fissa di 8 caratteri siffatte *HH:MM:SS* (HH: ora, MM: minuti, SS: secondi)

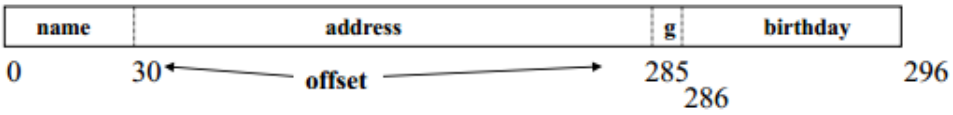
- **Tipi Enumerati**: sono i tipi *enum* che si possono creare. Si usa una codifica intera che associa in ordine ogni elemento a un intero.
  Esempio: week = {LUN,MAR,MER,...,DOM} vengono rappresentati come LUN: 00000001, MAR: 00000010, MER: 00000011, ...

2.2.2 Record a lunghezza fissa

Per ogni tipo di record nel database deve essere definito uno schema fisico che permetta di *interpretare correttamente il significato dei byte che costutuiscono il record*, cioè poter riuscire ad identificare i vari attributi all'interno di un record.

Figura 8:

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



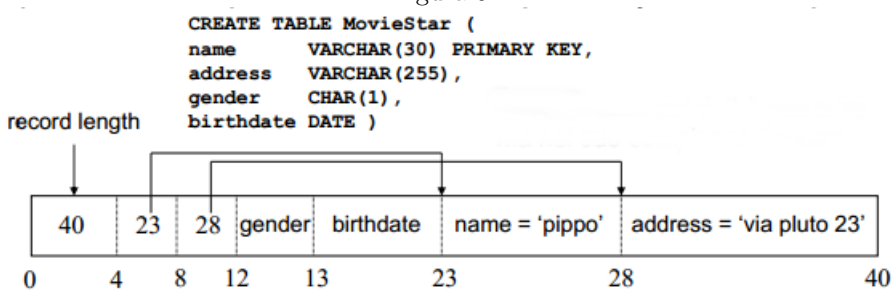
La situazione più semplice si ha quando evidentemente tutti i record hanno lunghezza fissa, in quanto, oltre alle informazioni logiche è sufficiente specificare l'ordine in cui gli attributi sono memorizzati nel record (se è differente dall'ordine di default), cioè se ho solo record a lunghezza fissa so esattamente come raggiungere un determinato campo, in quanto si sa a priori la lunghezza di ogni attributo, quindi partendo dal primo byte bisogna solo specificare l'offset per raggiungere l'attributo voluto.

2.2.3 Record a lunghezza variabile

Nel caso di record a lunghezza variabile si hanno diverse alternative che devono considerare anche i problemi legati aglia ggiornamenti che modificano la lunghezza dei campo e quindi dei record stessi.

Una soluzione consolidate consiste nel *memorizzare prima tutti i campi a lunghezza fissa e solo successivamente tutti i campi a lunghezza variabile*; per ogni campo a lunghezza variabile viene 'preposto' un *prefix pointer* che indica l'indirizzo del primo byte del campo. Dall'esempio in figura (Fig. 9) si può notare che il *record length* segna 40, poichè il record effettivamente è lungo 40 byte, quando in realtà la lunghezza dei dati è pari a soli 28 byte. Un ulteriore osservazione denota che non viene specificata la lunghezza dei record variabili poichè la si calcola facilmente come differenza dei *prefix pointer*.

Figura 9:



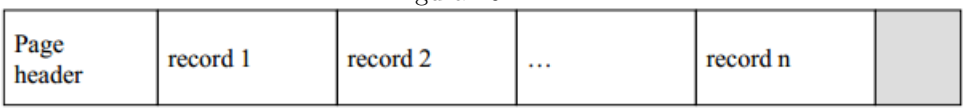
In generale ogni record oltre a contenere i dati, viene preposto da un **RECORD HEADER** (il cui formato varia a seconda del *dbms* usato) il quale, oltre a contenere la lunghezza del record, può contenere utili informazioni come:

- L'identificatore della relazione (tabella) a cui il record appartiene
- L'identificatore univoco del record nel database (RID **R**ecord **ID**entification o TID **T**uple **ID**entification)
- Un timestamp che indica quando il record è stato inserito o modificato l'ultima volta

2.2.4 Organizzazione Record in Pagine

Come già detto in precedenza nelle *pagine* vengono memorizzati i *record* appartenenti ad una stessa relazione.

Figura 10:



Normalmente la dimensione di un record è molto minore di quella di una pagina (anche se esistono tecniche particolari per gestire i *long tuples* la cui dimensione eccede quella di una pagina).

Nel caso di record a lunghezza fissa l'organizzazione di una pagina si potrebbe presentare come mostrato in figura (Fig. 10). In generale non posso utilizzare un'intera pagina per scrivere record a causa della presenza del **page header**.
Il **page header** mantiene informazioni quali:

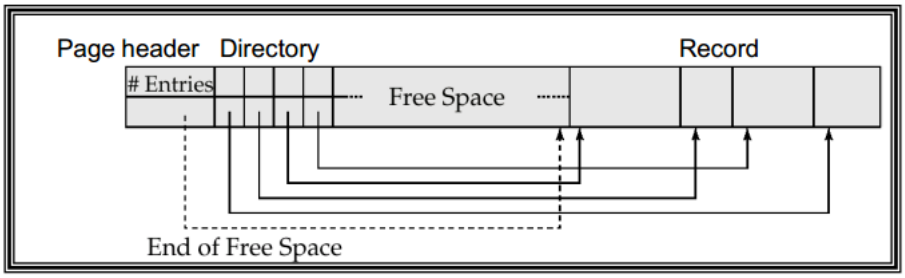
- ID della pagina nel database
- timestamp che indica quando la pagina è stata modificata l'ultima volta
- relazione (tabella) a cui appartengono i record (tuple) presenti nella pagina

Normalmente un record è contenuto interamente in una pagina dunque si può avere dello spazio sprecato.
Esempio:
Dimensione della pagina $P = 4KB = 4096byte$. Record di lunghezza fissa pari a 296 byte. Supponiamo l'header sia lungo 12byte, rimangono $4096 - 12 = 4084byte$ per i dati, pertanto è possibile memorizzare in una pagina fino a 13 record ($\lfloor \frac{4084}{296} \rfloor$), in ogni pagina resteranno quindi sempre inutilizzati almeno $4084 - 296 * 13 = 236$ byte. Se una fantasiosa relazione MovieStar contiene 10000 tuple serviranno quindi almeno 770 pagine ($\lceil \frac{10000}{13} \rceil$) per memorizzarla, e se per leggere una pagina da disco ci vogliono 10ms, la lettura di tutte le tuple richiederà circa 7.7 secondi (DECISAMENTE TROPPO!).

Con questo esempio si evince l'importanza di organizzare in maniera effeciente le tuple (record) all'interno di una pagina.

2.2.5 Organizzazione a slot delle pagine

Figura 11:

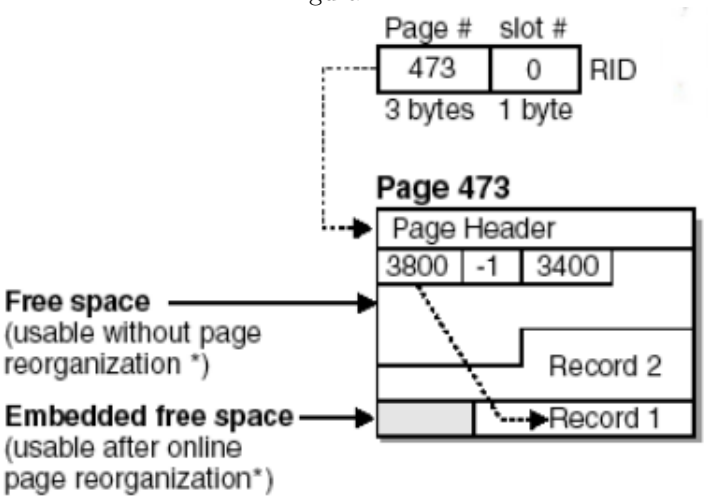


Il *formato tipico di una pagina in un dbms* è quello mostrato in figura (Fig. 11). È presente un page header già citato nella sezione precedente, seguita dalla *directory* e dai *record.* I **record** vengono memorizzati a partire dalla fine verso l'inizio, in maniera opposta alla **directory** che si espande dall'inizio verso la fine, questo per riuscire a controllare la collisione dello spazio di memoria occupato dalle due.

La *directory* contiene un puntatore per ogni record presente nella pagina, dunque cresce all'aumentare dei record. Con questa soluzione il **RID** (Record ID) è formato semplicemente dalla coppia (**PID**, **Slot**), nel quale il PID identifica la pagina in cui il *record* è memorizzato, mentre lo *slot* identifica il record all'interno della *directory*. Questo accoppiamento permette la *riallocazione* del record (ad esempio perchè si è modificato e richiede uno spazio in memoria più grande) nella pagina senza modificare il RID, inoltre permette di individuarlo velocemente.

Come viene indicato in figura esemplificativa (Fig. 12) il **RID** è solitamente di lunghezza 4bytes. I primi 3 bytes ne identificano il **PID** (dunque la pagina) e l'ultimo invece identifica lo **slot** (dunque lo spazio nella directory). L'esempio mostra un RID formato da PID 473 (che ne indica la pagina 473) e slot 0

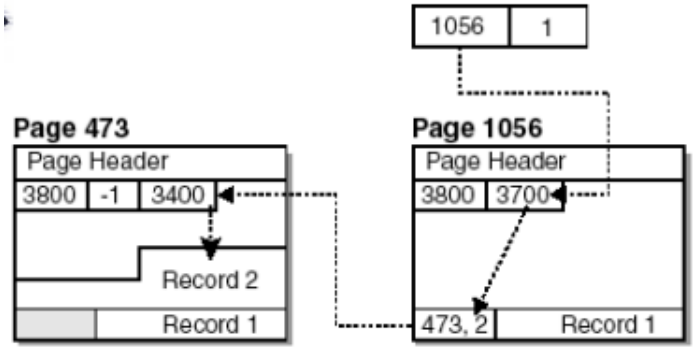
Figura 12:



(dunque il primo slot della directory), lo slot 0 della directory contiene l'indirizzo del record all'interno della pagina (ricorda che i record vengono memorizzati a partire dalla fine e andando "all'indietro"). Lo *spazio vuoto* è utilizzabile per l'allocazione di nuovi record, mentre è presente anche spazio un *embedded free space* che fino a che non viene riorganizzata la pagina, rimarrà perso, o al più disponibile per il record 1 o il record 2. N.B. Il -1 presente nel secondo slot della directory indica un record eliminato, che è quello che ha creato inoltre l'*embedded free space*.

Cosa succede se un record precedentemente memorizzato viene modificato e diventa troppo grande per riuscire ad essere riallocato all'interno della pagina stessa? Il record in questo caso viene spostato in un'altra pagina e si parla di **overflow**. Per evitare di aggiornare il **RID** del record (cosa per altro molto costosa poichè bisognrebbe modificare anche tutti gli indici in cui è presente questo record), esso rimane invariato e viene introdotto un livello di *indirezione*. Dunque al posto della memorizzazione del record stesso, nel RID del record troveremo un nuovo RID (coppia PID, SLOT) che ne indicherà la nuova pagina e il nuovo slot (vedi Fig. 13).

Figura 13:



Avere troppi RID overflow porta al degrado delle prestazioni, ecco che periodicamente è possibile richiedere la *riorganizzazione del file*.

2.3 Buffer Manager

La lettura di una tupla (record) richiede che la pagina in cui è salvata sia prima portata in memoria principale, in un'area gestita (esclusivamente) dal *dbms* detta **buffer pool** (ricorda che il *dbms* al momento dell'avvio si riserva automaticamente una parte di memoria proprio per gestire il buffer pool). Ogni *buffer* nel pool può ospitare una copia di una pagina su disco. La gestione del buffer pool, fondamentale dal punto di vista prestazionale, è demandata al **Buffer Manager (BM)**, già introdotto nel capitolo precedente. Il BM è chiamato in causa non solo per leggere le *pagine* ma anche per scrivere su disco una pagina che è stata modificata. Il BM ha inoltre un ruolo fondamentale nella gestione delle transazioni per garantire l'integrità del database a fronte di guasti. Proprio per questi motivi il BM non può essere "semplice" come quello previsto nei sistemi operativi, ma deve avere caratteristiche più specifiche per poter operare come indicato. *N.B. In DB2 si possono definire più buffer pool, ma ogni tablespace deve essere associato ad uno solo di essi.*

Come opera dunque il **Buffer Manager**?
A fronte di una *richiesta di una pagina*, segue la seguente procedura:

1. Se la pagina è già presente nel buffer, viene fornito al programma chiamante l'indirizzo del buffer in cui si trova la pagina.

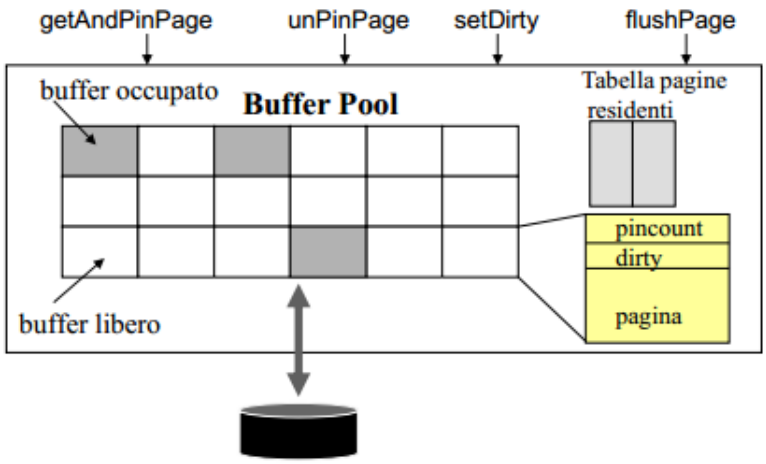
2. Se la pagina non è presente in memoria (dunque in nessun buffer):

   - Se c'è almeno un buffer libero, il BM seleziona uno fra questi e copia la pagina nel buffer selezionato.
   - Se non è presente alcun buffer libero allora il BM seleziona un buffer occupato da un'altra pagina, controlla che nessuno la stia usando (la pagina), in caso positivo la riscrive sul disco solo se è stata modificata e non è stata ancora salvata.

- A questo punto il BM può leggere la pagina e copiarla nel buffer prescelto (rimpiazzando, in caso fosse occupato, così quella prima presente).

2.3.1 Interfaccia del Buffer Manager

Figura 14:



L'interfaccia (cioè le azioni messe a disposizione) che il BM offre agli altri modubili del *dbms* ha quattro metodi base:

getAndPinPage richiede la pagina al BM e vi pone un *pin* ("spillo") ad indicarne l'uso. In caso la pagina sia presente restituisce direttamente l'indirizzo del buffer, altrimenti la carica come prima indicato. Si può immaginare dunque il bufferpool come una bacheca con dei fogli, ogni volta che chiedo di leggere un foglio se questo non è presente nella bacheca, viene appeso il foglio e messo uno spillo per tenerlo attaccato, altrimenti se il foglio è già presente dò indicazioni, mettendo uno spillo a mia volta, che sto controllando la pagina. [Azione: +1 pincount]

unPinPage rilascia la pagina ed elimina un pin. Quando il programma termina di utilizzare la pagina richiesta col metodo precedente, semplicemente toglie uno "spillo" alla pagina. In caso il numero di spilli sia uguale a 0, il *dbms* segna questa pagina come *potenzialmente* sostituibile in caso ci sia bisogno di un buffer per caricare una nuova pagina. [Azione: -1 pincount]

setDirty indica che la pagina è stata modifica, ovvero è **dirty** (è stata resa "sporca"). Quando un programma che ha richiesto una pagina la utilizza e la modifica, deve indicare al *dbms* che la pagina non è più uguale a quella che è presente sul disco in memora secondaria. Questo approccio è molto performante, in quanto il *dbms* non deve ogni volta scrivere le modifiche

di una pagina sul disco, tanto per poter accedere a una pagina bisogna sempre passare per il **BM**, dunque le modifiche permangono sulla memoria centrale e solo di tanto in tanto esse vengono realmente "aggiornate" sul disco. [Azione: pone a 1 il bit *dirty*]

flushPage forza la scrittura della pagina su disco, rendendola così pulita. [Azione: pone a 0 il bit *dirty*]

Come inoltre si vede dall'immagine (Figura 14) in realtà le pagine all'interno del bufferpool contengono anche due informazioni aggiuntive: il campo **pincount** (è un valore numerico intero non negativo) e il bit **dirty**.

2.3.2 Politiche di rimpiazzamento

Nei sistemi operativi una comune politica adottata per decidere quale pagina rimpiazzare è la **LRU** (Least Recently Used), ovvero *viene rimpiazzata la pagina che da più tempo non è in uso.* Per adottare una politica di questo tipo nei *dbms*, è necessario mantenere un'informazione sull'ultimo utilizzo effetuato sulla *pagina.* Nei *dbms* LRU non sempre è una buona scelta, in quanto per alcune query il "pattern d'accesso" ai dati è noto, e può quindi essere utilizzato per operare scelte più accurate, in grado di migliorare anche di molto le prestazioni.

L'**hit ratio**, ovvero la frazione di richieste che non provocano una operazione I/O (cioè le richieste che vanno "a segno" senza dover andar a chiedere la pagina al disco), indica sinteticamente quonto buona è una politica di rimpiazzamento. Un esempio, come vedremo, riguarda algoritimi di join che scandiscono N volte le tuple di una relazione; in questo caso la politica migliore da utilizzare sarebbe la **MRU** (Most Recently Used), ovvero rimpiazzare la pagina usata più recentemente.

Tutto quello appena detto dovrebbe darci un ulteriore motivo per comprendere per cui i *dbms* non usino (tutti) i serizi offerti dai sistemi operativi.

2.4 Organizzazione dei file

Il modo in cui i *record* (tuple) vengono organizzati nei file incide sull'efficienza delle operazioni e sull'occupazione di memoria.

Per semplicità di calcoli considereremo *record a lunghezza fissa* e valuteremo i "costi" come *numero di operazioni I/O*, assumendo che ogni richiesta di una pagina comporti un'operazione di I/O e che ogni operazione effettuata sulle pagine già caricate in memoria abbia costo zero.

Siamo interessati a stimare il costo delle seguenti operazioni:

- Ricerca per chiave (WHERE column_name=value1)
- Ricerca per intervallo (WHERE column_name BETWEEN value1 AND value2)
- Inserimento di un nuovo record

- Cancellazione di un record
- Modifica del valore di un attribo di un record (sia chiave (UNIQUE) che non chiave (Valori ripetuti))

Assumeremo inoltre come costo di base il numero di accessi alla memoria secondaria, ipotesi semplificativa poichè non viene preso in considerazione il fatto che gli accessi possano essere sequenziali o meno.

Abbiamo bisogno di alcune informazioni per poter valutare i costi, informazioni reperibili attraverso i **cataloghi**.

Ogni *dbms* mantiene dei **cataloghi**, ovvero delle *relazioni che descrivono il DB sia a livello logico che fisico.* I cataloghi che ci interessano in questa fase sono quelli che riportano *informazioni statistiche sulle relazioni*, in particolare

Tabella 1:

SQL Catalog	SQL Attribute	Descrizione	SIMBOLO
SYSSTAT.TABLES	CARD	Numero di tuple nella relazione	NR
SYSSTAT.TABLES	NPAGES	Numero di pagine occupate dalla relazione	NP
SYSSTAT.COLUMNS	COLCARD	Numero di valori distinti dell'attributo	NK
SYSSTAT.COLUMNS	LOW2KEY	Secondo valore minore	LK
SYSSTAT.COLUMNS	HIGH2KEY	Secondo valore maggiore	HK

SYSSTAT.TABLES contiene informazioni riguardanti le tabelle.

SYSSTAT.COLUMNS contiene invece informazioni per ogni colonna di ogni tabella (dunque la chiave identificativa in questo catalogo è la coppia <nometabella,nomeattributo>), di particolare interesse per il nostro scopo, guarderemo gli attributi *COLCARD*, *LOW2KEY*, *HIGH2KEY*.

COLCARD indica il numero di valori diversi per il determinato attributo. Poichè è un calcolo oneroso e difficile, in realtà COLCARD restituisce una stima sul vero valore, anche perchè è un attributo che viene aggiornato "di tanto in tanto".

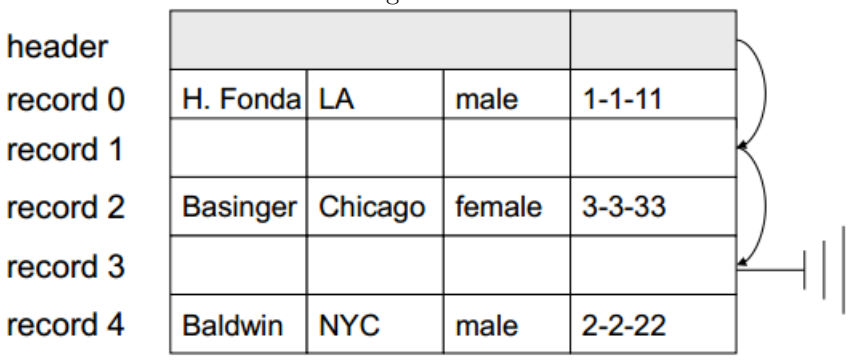
LOW2KEY/HIGH2KEY indicano il secondo valore minore e il secondo valore maggiore rispettivamente. Viene indicato il secondo e non il primo poichè a volte il progettista utilizza i valori massimi e/o minimi come valori 'warning' invece del valore *null* (a causa anche del fatto che il null può avere significati diversi a seconda dell'utilizzo: valore sconosciuto, valore non applicabile, valore né conosciuto né applicabile). Esempio: utilizzare il valore -1 nell'attributo Età per indicare che non si hanno informazioni riguardo l'età per una determinata persona. Questi attributi sono utili per riuscire a sapere l'intervallo dei valori dell'attributo.

2.4.1 Heap File VS Sequential File

HEAP FILE Detta anche organizzazione seriale, è la più semplice in quanto si caratterizza per *l'inserimento di nuovi record alla fine del file.* Se qualche

record viene cancellato, per poter riutilizzare lo spazio (ricorda che stiamo parlando di record a lunghezza fissa, quindi riutilizzabili senza accorgimenti sulla grandezza) senza dover scandire tutto il file, è necessario implementare un meccanismo per localizzare velocemente gli spazi liberi.

Figura 15:



Vedremo la gestione attraverso **linked list**, **directory** e l'implementazione attuata in **DB2**.

Linked List La prima possibilità è quella di mantenere due *liste doppiamente concatenate* (cioè scandibili da entrambi i lati):

- una lista per le pagine piene (completamente)
- una lista per le pagine con spazio vuoto

Tipicamente, con record di lunghezza variabile, quasi tutte le pagine avranno spazio vuoto e dunque per trovare una pagina in grado di contenere un nuovo record potrebbe essere necessario scandire tutta la lista.

Directory Un'altra possibilità è quella di mantenere una *directory* di pagine: la *directory* stessa è organizzata in pagine collegate a lista. Ogni *entry* della directory identifica una pagina del file e può contere anche lo spazio libero presente, di conseguenza per trovare una pagina per contenere un nuovo record basta scandire la *directory* (tipicamente molto più piccola del file). A vantaggio della performance però si deve pagare una dimensione maggiore del file (dovuta alla presenza della *directory* stessa).

Implementazione DB2 Ricordiamo che DB2 raggruppa le pagine in *extent*. Una pagina ogni 500 contiene un **Free Space Control Record** (FSCR) contenente una directory dello spazio libero nelle successive 500 pagine (fino al prossimo FSCR). Questo vuol dire che per leggere l' "intera" *directory* bisogna saltare di 500 pagine in 500 e leggere i vari FSCR.

La *dimensione della pagina* (4/8/16/32 KB) può essere specificata all'atto della creazione del *tablespace*. Dimensioni maggiori implicano accessi sequenziali ma con rischio di frammentazione, dimensioni minori utili per accessi random.

PRESTAZIONI La tabella (Tab. 2) riassume i costi per le operazioni di base.

Tabella 2:

Operazione	**Descrizione**	**Costo**
Ricerca per chiave	La ricerca avviene leggendo sequenzialmente le pagine	$\frac{NP}{2}$ medio, NP massimo e se non presente
Ricerca per intervallo	Devono essere lette per forza tutte le pagine	NP sempre
Inserimento	Si assume di inserire in fondo al file	2
Cancellazione	Si assume di cancellare un record	C(ricerca)+1
Aggiornamento	Si assume di aggiornare un record	C(ricerca)+1

La ricerca per chiave ha costo massimo NP in quanto se il *record* si trova nell'ultima pagina (o non è presente) devo scandirle tutte; per analogia la ricerca per intervallo richiede di scandire per forza tutte le pagine poichè non so a priori se fino all'ultima esiste un record che abbia un valore compreso nell'intervallo. L'inserimento ha costo 2 (ricordiamo che il costo è dato da una operazione IO) poichè prima devo cercare una pagina che abbia spazio per il record (1) e poi caricarla dal disco in memoria centrale (2). Infine la cancellazione e aggiormento richiedono un costo *C(ricerca)+1* dove il C(ricerca) è il costo della ricerca che viene utilizzata per cancellare e/o aggiornare.

SEQUENTIAL FILE In un file sequenziale i *record* (tuple) vengono mantenuti ordinati secondo i valori di un attributo o di una combinazione di attributi. È evidente che gli inserimenti devono ora avvenire in maniera ordinata (quindi il costo sarà maggiore), ma le ricerche beneficeranno dell'ordinamento (si potrà usare la ricerca binaria): normalmente per tentare di migliorare il costo dell'inserimento vengono lasciati spazi liberi in ogni pagina (Fig. 16) oppure vengono tollerati *record overflow* con una successiva riorganizzazione.

Figura 16:

Brighton	A-127	750
Downtown	A-101	500
Downtown	A-101	600
Mianus	A-215	700
Perryridge	A-102	400
Perryridge	A-201	900

N.B. Non è richiesto che siano ordinate le singole tuple all'interno di una stessa pagina, ma bensì che ci sia un ordine totale nelle pagine.

PRESTAZIONI La tabella (Tab. 3) riassume i costi per le operazioni di base. Per semplicità si considera che il file sia ordinato sui valori di chiave primaria (o di una chiave candidata).

Tabella 3:

Operazione	**Descrizione**	**Costo**
Ricerca per chiave	Si utilizza la ricerca binaria per trovare la pagina che contiene il record	$\lceil \log_2 NP \rceil$
Ricerca per intervallo	Si leggono le sole pagine con valori di chiave nell'intervallo [L,H]	C(ricerca)-1+$\frac{(H-L)*NP}{HK-LK}$
Inserimento	Si suppone che vi sia spazio per l'inserimento	C(ricerca)+1
Cancellazione	Si assume di cancellare un record	C(ricerca)+1
Aggiornamento	Si assume di aggiornare un record	C(ricerca)+1

Per la ricerca per chiave si applica una semplice *ricerca binaria*. Per la ricerca per intervallo si fa un ragionamento equivalente, il costo è dovuto al trovare la pagina che contiene il primo valore dell'intervallo per poi leggere in sequenza le pagine fino a trovare l'ultimo record di interesse nell'intervallo. Infine si nota come per l'inserimento ora il costo non risulti più di 2, ma bensì in base alla ricerca effettuata per inserire il dato. Come descritto nel paragrafo precedente, *C(ricerca)* identifica il costo della ricerca effettuata per inserimento/cancellazione/aggiornamento.