

# Sistemi Informativi – Lezione 2

---

Prof. Paolo Ciaccia  
pciaccia@deis.unibo.it

DEIS – Università degli Studi di Bologna

# Argomenti della lezione

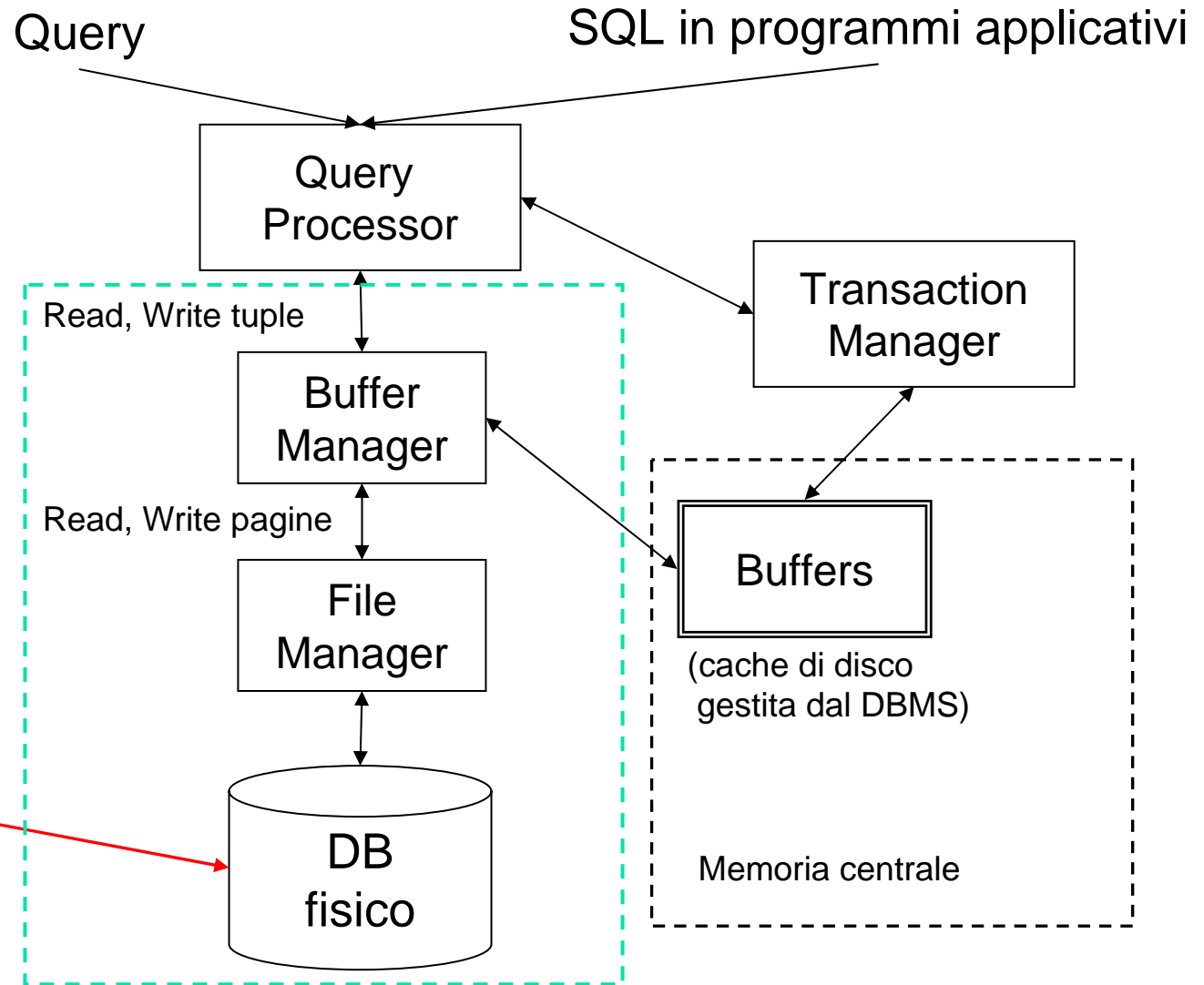
---

- Il DB fisico
- Gestione delle transazioni
- Elaborazione di interrogazioni
- DB attivi: trigger e regole
- Estendibilità

## II DB fisico

---

# Parte dell'architettura di un DBMS



Il DB logico è fisicamente memorizzato su dispositivi di memoria permanente

# Pagine

---

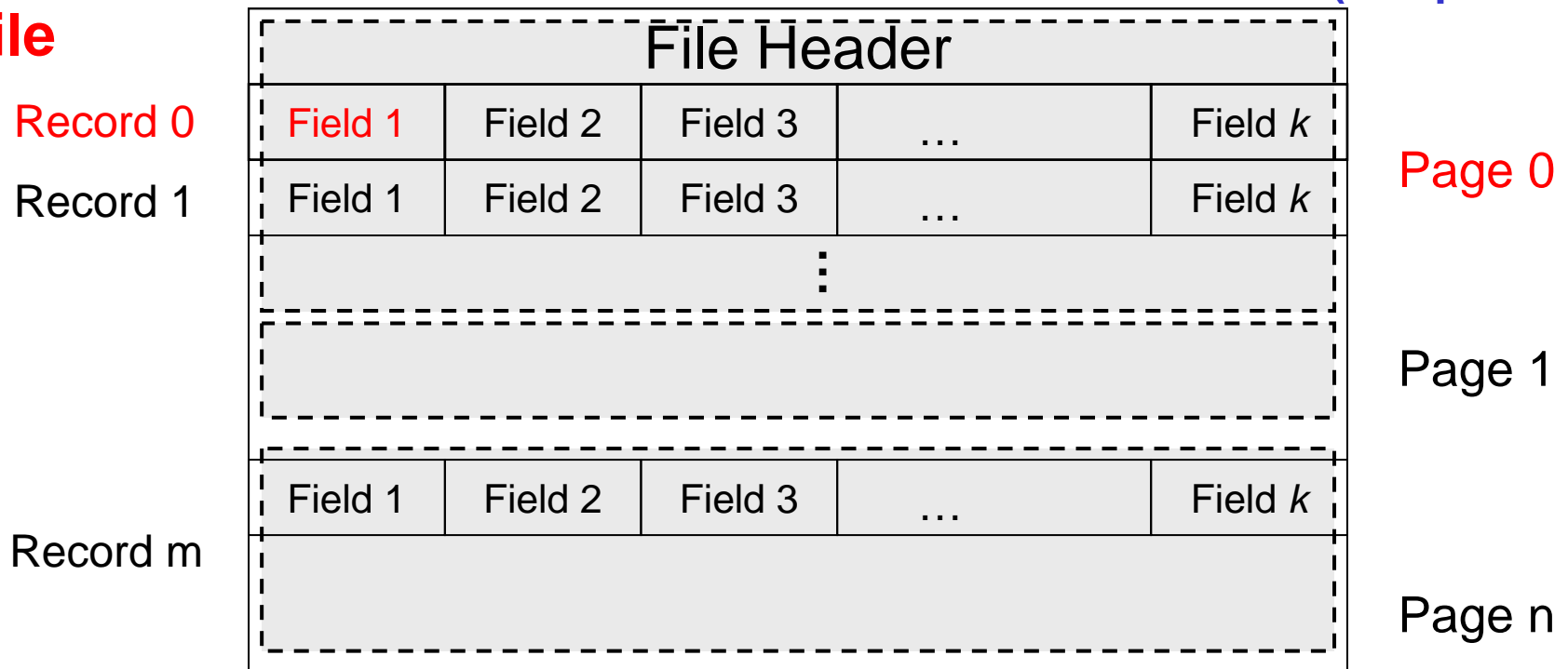
- Un DB, a causa della sua dimensione, risiede normalmente su dischi (e eventualmente anche su altri tipi di dispositivi), quindi i dati devono essere trasferiti in memoria centrale per essere elaborati dal DBMS
- Il trasferimento non avviene in termini di singole tuple, bensì di blocchi (o pagine, termine comunemente usato quando i dati sono in memoria)
- **Pagina**: sequenza contigua di byte su disco, di dimensione tipica pari a qualche KB (4 - 64 KB)
  - 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
- Il tempo necessario a leggere una pagina dipende dalla dimensione della pagina e da caratteristiche del dispositivo, e tipicamente ha valori dell'ordine della decina di millisecondi

# File, pagine, record e campi

- A livello fisico un DB consiste di un insieme di file, ognuno dei quali è una collezione di pagine, di dimensione fissa (es: 8 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

## Schema di riferimento (semplificato)

### File



# File: una precisazione

---

- I “file” del DBMS non corrispondono necessariamente a quelli del file system del sistema operativo
- Alcuni casi possibili (in pratica ogni DBMS adotta soluzioni specifiche):
  - Ogni relazione è memorizzata in un proprio file
  - Tutto il DB è memorizzato in un singolo file
- I riferimenti seguenti sono tratti, rispettivamente, da:
  - PostgreSQL 8.2.0 Documentation
  - MySQL 5.0 Reference Manual

# Rappresentazione dei valori

---

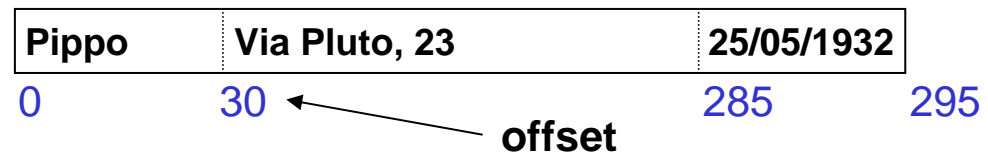
- Per ogni tipo di dati di SQL viene definito un formato di rappresentazione, ad es.:
- Stringhe a lunghezza fissa: **CHAR(n)**
  - Si usano n byte, eventualmente usando un carattere speciale per valori lunghi meno di nEsempio: se A è CHAR(5), 'cat' viene memorizzato come cat␣␣
- Stringhe a lunghezza variabile: **VARCHAR(n)**
  - Si allocano m byte per i caratteri effettivamente presenti + altri per memorizzare il valore di mEsempio: se A è VARCHAR(10), 'cat' viene memorizzato in 4 byte come 3cat



# Schema fisico dei record

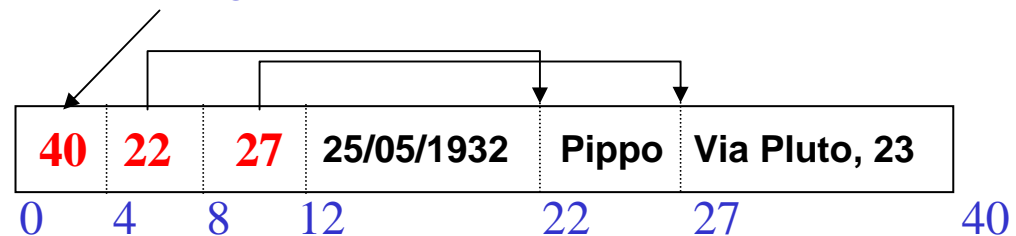
- Per ogni tipo di record nel DB è definito uno **schema (fisico)** che permette di interpretare correttamente il significato dei byte che lo costituiscono
- **Record a lunghezza fissa**: basta specificare l'ordine in cui gli attributi sono memorizzati nel record (se differente da quello di default)

```
CREATE TABLE MovieStar (  
name CHAR(30),  
address CHAR(255),  
birthdate DATE )
```



- **Record a lunghezza variabile**: diverse alternative, dati i problemi legati agli aggiornamenti che possono modificare la lunghezza dei campi
- Esempio: **prima i campi a lunghezza fissa, quindi quelli a lunghezza variabile**, per ognuno dei quali si ha un **puntatore**

```
CREATE TABLE MovieStar ( record length  
name VARCHAR(30),  
address VARCHAR(255),  
birthdate DATE )
```



# Record Header

---

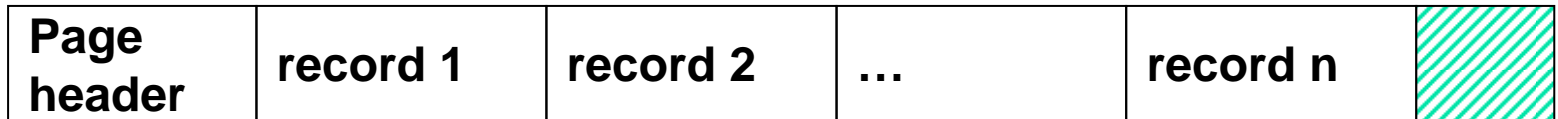
- In generale ogni record include anche un **header** che, oltre alla lunghezza del record, può contenere:
  - L'identificatore della relazione cui il record appartiene
  - L'identificatore univoco del record nel DB, o **RID**
  - Un timestamp, che indica quando il record è stato inserito o modificato l'ultima volta
- Il formato specifico dell'header ovviamente varia da un DBMS all'altro

# Organizzare i record in pagine

---

- Normalmente la dimensione di un record è minore di quella di una pagina
  - Esistono tecniche particolari per gestire il caso di “long tuples”, la cui dimensione eccede quella di una pagina (es. TOAST in PostgreSQL)

- Concettualmente, l'organizzazione in una pagina si potrebbe presentare così:



in cui il **page header** mantiene informazioni quali l'ID della pagina nel DB, o **PID**, un timestamp che indica quando la pagina è stata modificata l'ultima volta, ecc.

- Per **record a lunghezza variabile**, un update può far aumentare la dimensione di un record e, se non c'è più spazio nella pagina, forzarne lo spostamento in un'altra pagina ( “**overflow**”)
- Avere molti record in overflow porta a un degrado delle prestazioni, per cui si rende necessario **riorganizzare periodicamente il file**

# Organizzazione dei file

---

- Il modo con cui i record vengono organizzati nei file incide sull'efficienza delle operazioni e sull'occupazione di memoria
- Nel seguito vediamo alcune organizzazioni di base, ovvero:

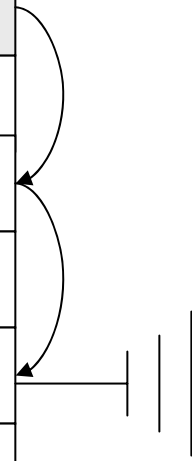
Heap file, Sequential file, Hash file

considerando per semplicità record a lunghezza fissa

# 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 senza dover scandire tutto il file, è necessario implementare un meccanismo per localizzare velocemente gli spazi liberi (ad es. lista)
- Prestazioni scadenti per operazioni di ricerca, ottime per inserimento

header				
record 0	H. Fonda	LA	male	1-1-11
record 1				
record 2	Basinger	Chicago	female	3-3-33
record 3				
record 4	Baldwin	NYC	male	2-2-22



The diagram illustrates a heap file structure. It consists of a header row followed by five record rows. The records are: record 0 (H. Fonda, LA, male, 1-1-11), record 1 (empty), record 2 (Basinger, Chicago, female, 3-3-33), record 3 (empty), and record 4 (Baldwin, NYC, male, 2-2-22). A pointer mechanism is shown on the right side, with a vertical line and a horizontal line pointing to the right. Two curved arrows indicate the flow of data: one from record 0 to record 2, and another from record 2 to record 4, suggesting a linked list structure for non-sequential records.

# Sequential file

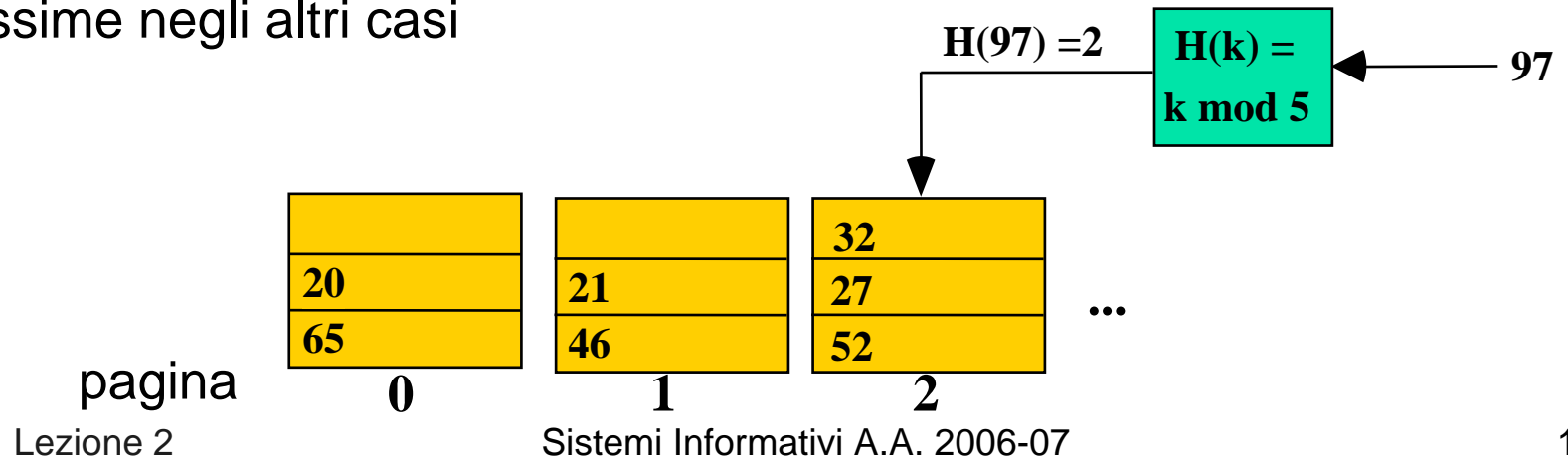
---

- In un file sequenziale i record vengono mantenuti ordinati secondo i valori di un attributo (o di una combinazione di attributi)
- È evidente che gli inserimenti devono ora avvenire ordinatamente, e quindi normalmente viene lasciato dello spazio libero in ogni pagina (oppure si tollerano record in overflow e poi si riorganizza)
- Prestazioni discrete per ricerche sull'attributo di ordinamento, scadenti negli altri casi

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

# Hash file

- In un file hash i record vengono allocati in una pagina il cui indirizzo dipende dal valore di chiave del record:  
**key  $\rightarrow$  H(key)  $\rightarrow$  page address**
- Una comune funzione hash è il resto della divisione intera:  
**H(k) = k mod NP**
- Si può applicare anche a chiavi alfanumeriche dopo averle convertite
- Anche in questo caso è necessario prevedere la possibilità di record in overflow
- Prestazioni ottime per ricerche sull'attributo dominio della f. hash, pessime negli altri casi



# Indici

---



# Perché gli indici

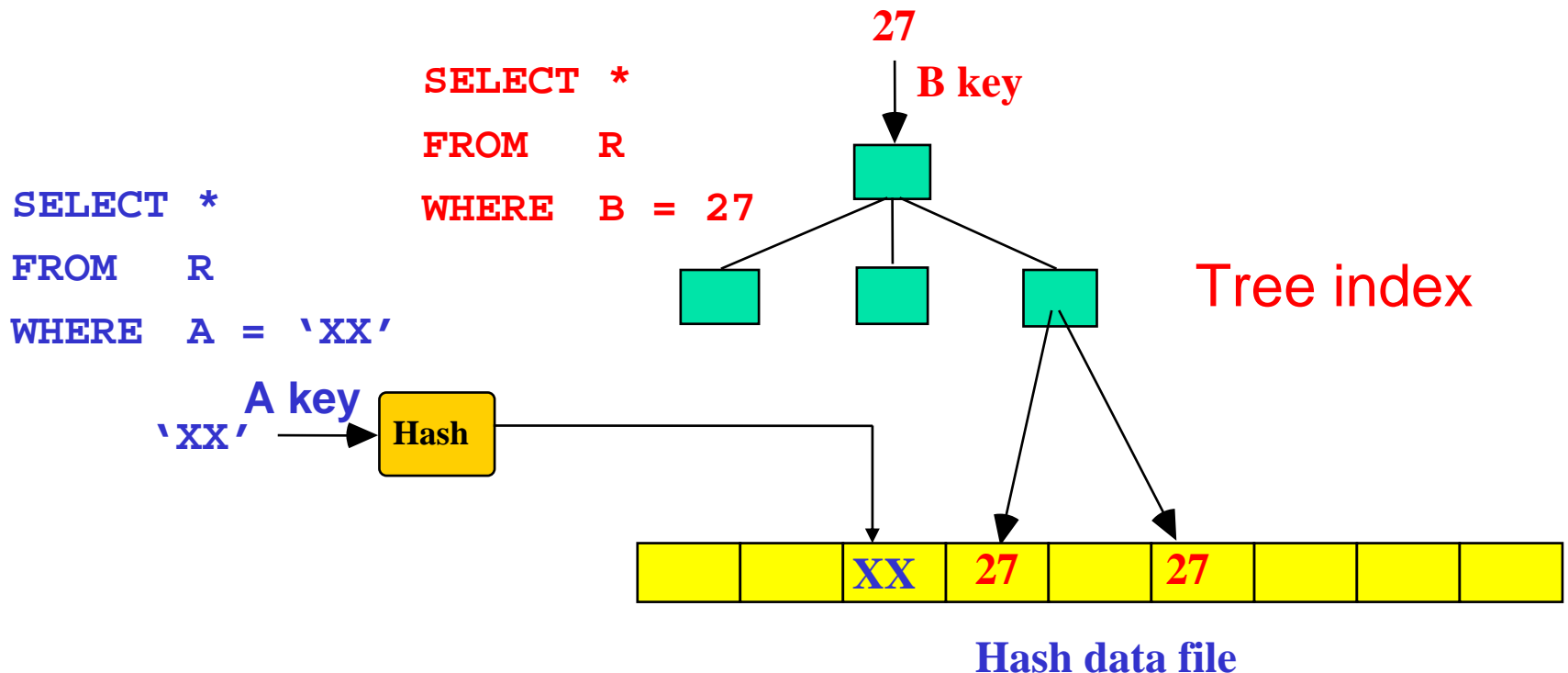
---

- Le organizzazioni dei file viste (heap, sequenziale, hash) non permettono di ottenere prestazioni soddisfacenti se si eseguono ricerche su attributi che non sono usati per determinare il criterio di allocazione dei record
- Anche nel caso di file sequenziali, una ricerca per valore di chiave richiede comunque un numero di operazioni di I/O pari a  $\lceil \log_2 NP \rceil$ , con NP numero di pagine del file, e quindi elevato per file di grandi dimensioni
- Per ovviare a questi limiti si creano degli indici che forniscono “cammini di accesso” alternativi ai dati

# Cammini di accesso

- La costruzione di indici su una relazione mette a disposizione modalità alternative (**cammini di accesso**) per localizzare velocemente i dati di interesse

**NB** Si usa comunemente il termine (valore di) chiave (di ricerca) per indicare il valore di un campo usato per selezionare i record (es. B è una chiave)



# Indici: principio di base

- Logicamente, un indice può essere visto come un insieme di coppie del tipo  $(k_i, p_i)$  dove:
  - $k_i$  è un **valore di chiave** del campo su cui l'indice è costruito
  - $p_i$  è un **puntatore al record** (eventualmente il solo) con valore di chiave  $k_i$ . Nei DBMS è quindi un **RID**
- I diversi indici differiscono nel modo con cui organizzano l'insieme di coppie  $(k_i, p_i)$ , problema importante in quanto, benché un indice occupi minor spazio rispetto al file dati, può raggiungere notevoli dimensioni che determinano **problemi di gestione simili a quelli del file dati**

File dati

RID	CodImp	Nome	Sede	Ruolo	Stipendio
(27,1)	E005	Neri	S02	Analista	2500
<b>(27,2)</b>	E003	Bianchi	S01	Programmatore	1000
(30,1)	E004	Gialli	S03	Programmatore	1000
(30,2)	E001	Rossi	S01	Analista	2000
(30,3)	E002	Verdi	S02	Sistemista	1500

Indice su CodImp

CodImp	RID
E001	(30,2)
E002	(30,3)
<b>E003</b>	<b>(27,2)</b>
E004	(30,1)
E005	(27,1)

# Tipi di indici

---

- Esistono diverse tipologie di indici; la prima distinzione è tra:
  - **Indici ad albero**: i valori di chiave  $k_i$  vengono mantenuti ordinati e organizzati in una struttura ad albero, in modo da poter essere reperiti più efficientemente
  - **Indici hash**: si usa una funzione hash per determinare la posizione dei valori di chiave  $k_i$  nelle pagine dell'indice
    - questi indici forniscono prestazioni eccellenti per ricerche di uguaglianza (es.  $A = 5$ ), tuttavia non possono essere usati per ricerche di intervallo (es.  $A \text{ BETWEEN } 5 \text{ AND } 20$ )
- Ulteriori distinzioni si hanno considerando
  - **Indici mono-** (es. **B-tree**) o **multi-dimensionali** (es. **R-tree**)
  - **Indici funzionali**, **metrici**, ecc.

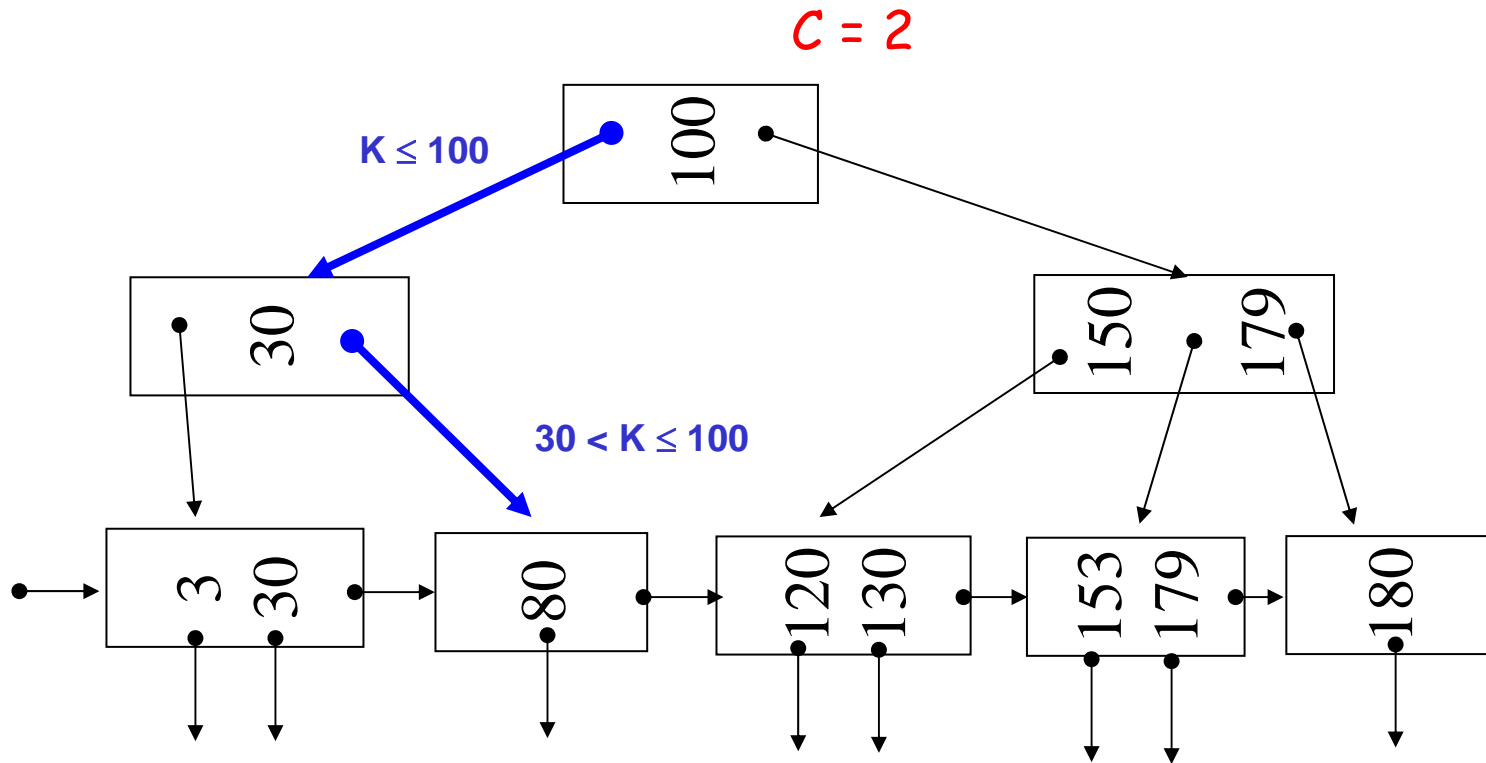
# B<sup>+</sup>-tree

---

- Il B<sup>+</sup>-tree è la **struttura comunemente usata nei DBMS per realizzare indici ad albero dinamici**, ossia che non richiedono riorganizzazioni periodiche
  - E' una **variante del B-tree** (Bayer McCreight, 72), nome che spesso viene impropriamente usato per riferirsi ai B<sup>+</sup>-tree
- Principali caratteristiche:
  - Tutti le coppie  $(k_i, p_i)$  sono contenute in pagine (o **nodi**) “**foglia**”
    - Le foglie sono collegate a lista mediante puntatori (PID) per favorire la risoluzione di query di intervallo
  - Ogni **foglia** contiene un **numero di coppie che varia tra  $c$  e  $C$** , con  **$C = 2c$** , ovvero è piena almeno al 50%
  - Ogni **nodo intermedio** (né radice né foglia) contiene **da  $c$  a  $C$  valori di chiave** che fungono da “**separatori**” e un numero di figli variabile **tra  $c+1$  e  $C+1$  figli**, ovvero è pieno almeno al 50%
  - La **radice**, se non è una foglia, ha **almeno 2 figli**
  - Ogni percorso dalla radice a una foglia ha lunghezza  **$h$**  (altezza del B<sup>+</sup>-tree), quindi l'albero è perfettamente bilanciato

# Esempio di B<sup>+</sup>-tree

- La figura mostra un esempio di B<sup>+</sup>-tree di altezza h = 3



# Ordine e altezza

---

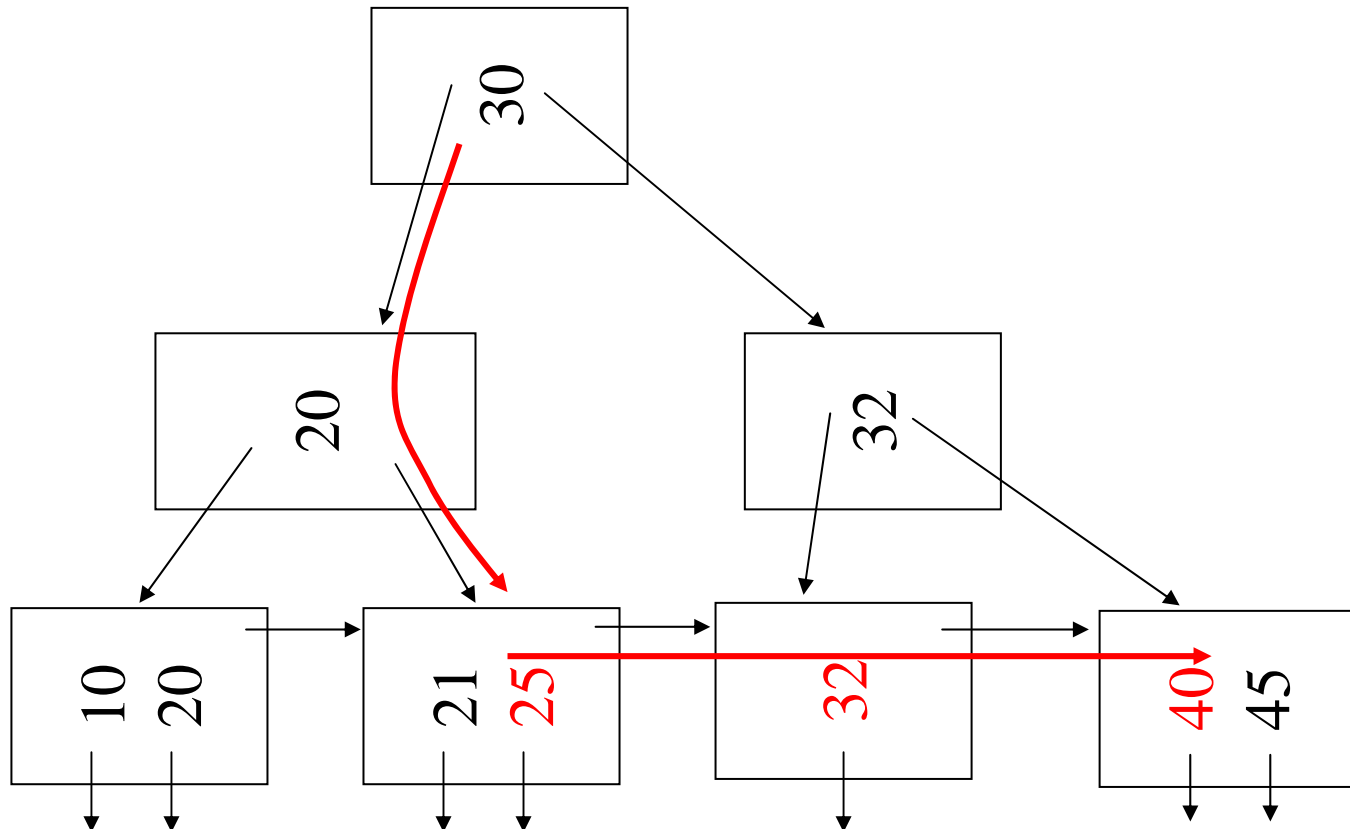
- L'altezza  $h$  è una funzione logaritmica di  $NK$  (numero di valori di chiave indicizzati); in particolare si può dimostrare che vale la relazione

$$1 + \left\lceil \log_{c+1} \frac{NK}{C} \right\rceil \leq h \leq 2 + \left\lfloor \log_{c+1} \frac{NK}{C} \right\rfloor$$

- Per avere un'idea del valore di  $c$ , con valori di chiave di 8 byte, RID di 4 byte e pagine con disponibili 4000 byte si ottiene  $c = 166$
- Pertanto se  $NK = 10^9$ , la ricerca di un valore di chiave (che accede a un nodo per ogni livello) richiede al massimo:  
 $2 + \lfloor \log_{167} (10^9 / 332) \rfloor = 4$  operazioni di I/O!
  - Per contro, una ricerca binaria richiederebbe 22 accessi a disco, supponendo di avere le pagine indice piene
- La variabilità di  $h$  è, fissati  $NK$  e  $c$ , molto limitata (differenza di 1 tra minima e massima)
- Con questi valori di  $c$ , con  $h = 3$  si gestiscono fino a circa 3 milioni e mezzo di chiavi

# Ricerche per intervallo

- Per cercare tutti i valori di chiave nell'intervallo  $[L,H]$  si cerca innanzitutto il valore  $L$ , quindi si prosegue in sequenza sulle foglie  
Esempio: si cercano le chiavi nell'intervallo  $[23,42]$





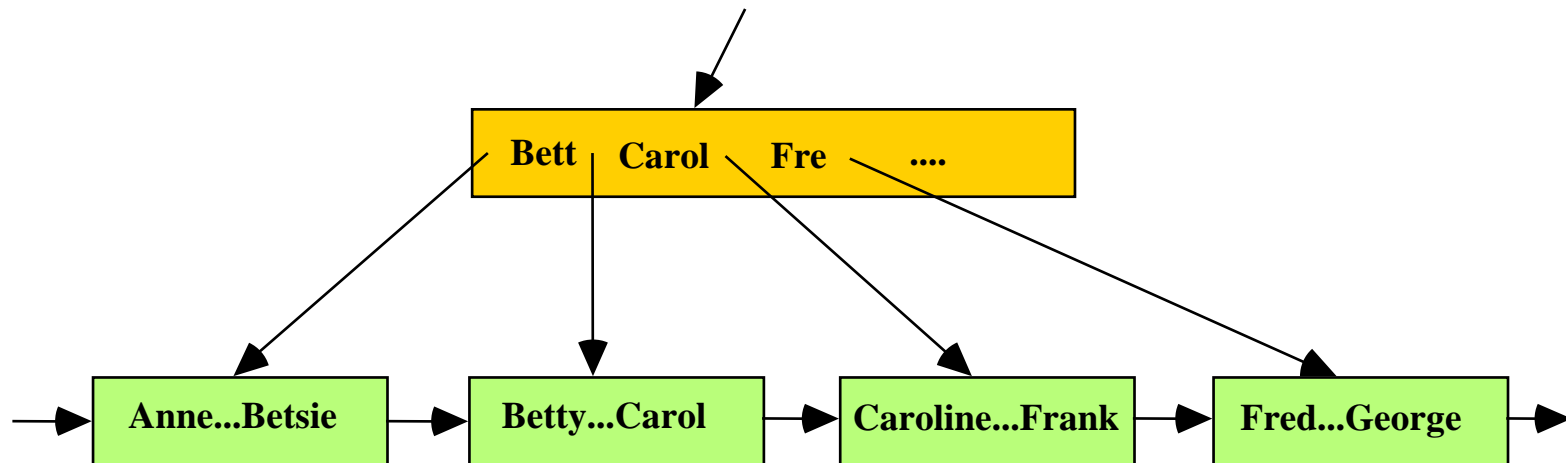
# Inserimento

---

- La procedura di inserimento procede innanzitutto **cercando la foglia in cui inserire il nuovo valore di chiave**
- **Se c'è posto** la nuova coppia  $(k_i, p_i)$  viene inserita nella foglia e la procedura termina
- **Se non c'è più posto**, si attiva una **procedura di “split” ricorsiva** (qui non descritta) che, al limite, si propaga fino alla radice
- La procedura di cancellazione può seguire una logica simile, ma in pratica spesso si procede a una riorganizzazione periodica (***reindexing***)

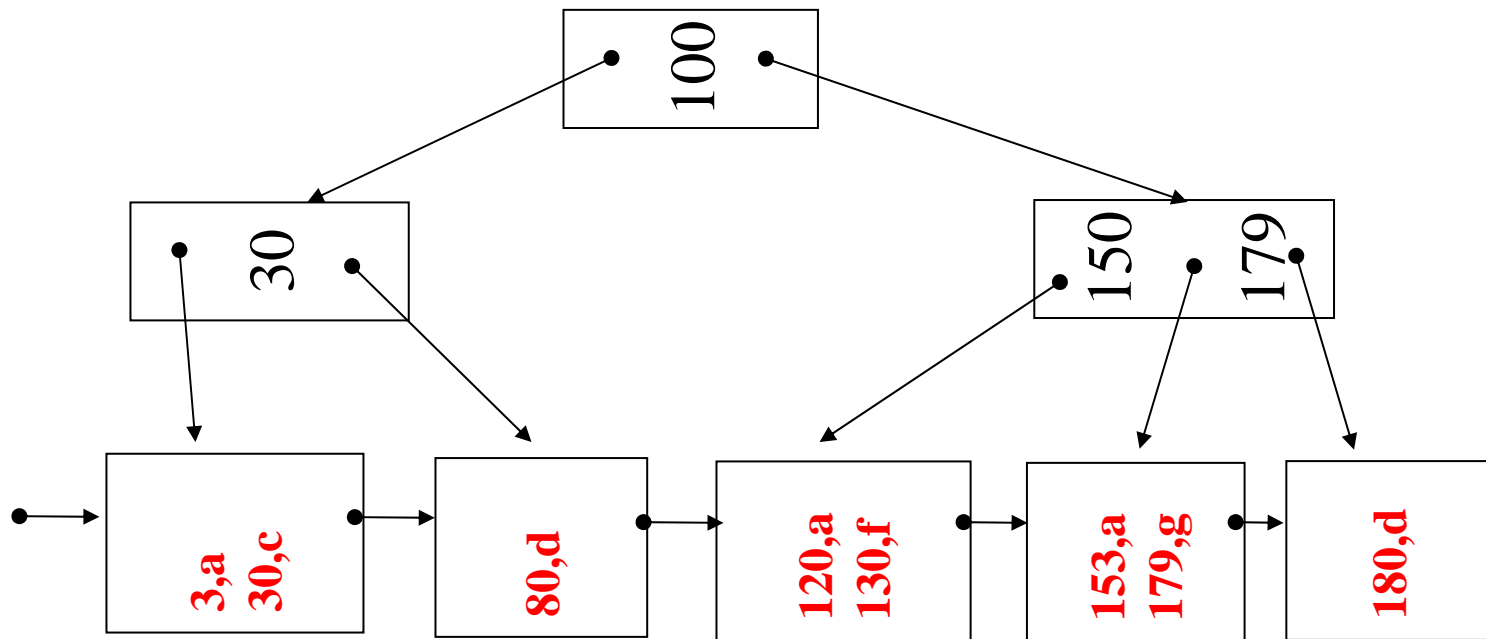
# B+-tree per chiavi a lunghezza variabile

- Nel caso di chiavi a lunghezza variabile, la capacità di un nodo non può essere più definita in modo esatto, ma mantiene un significato solamente in senso medio
- D'altronde con chiavi a lunghezza variabile è possibile comprimere i separatori presenti nei nodi interni (non foglia)  
Esempio: Per separare "Frank" da "Fred" è sufficiente la stringa "Fre"



# B+-tree per l'organizzazione dei dati

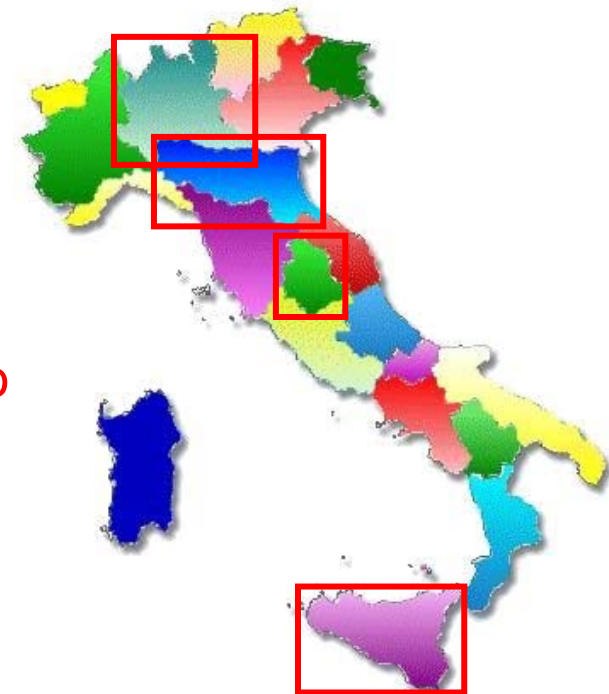
- In alcuni casi, un B+-tree può essere usato per organizzare dinamicamente i dati stessi (es. table InnoDB di MySQL)
- Nelle foglie, anziché mantenere coppie  $(k_i, p_i)$ , si memorizzano i record dati completi
- In questo caso si parla anche di **indice "clustered"** (ma la terminologia non è standard)



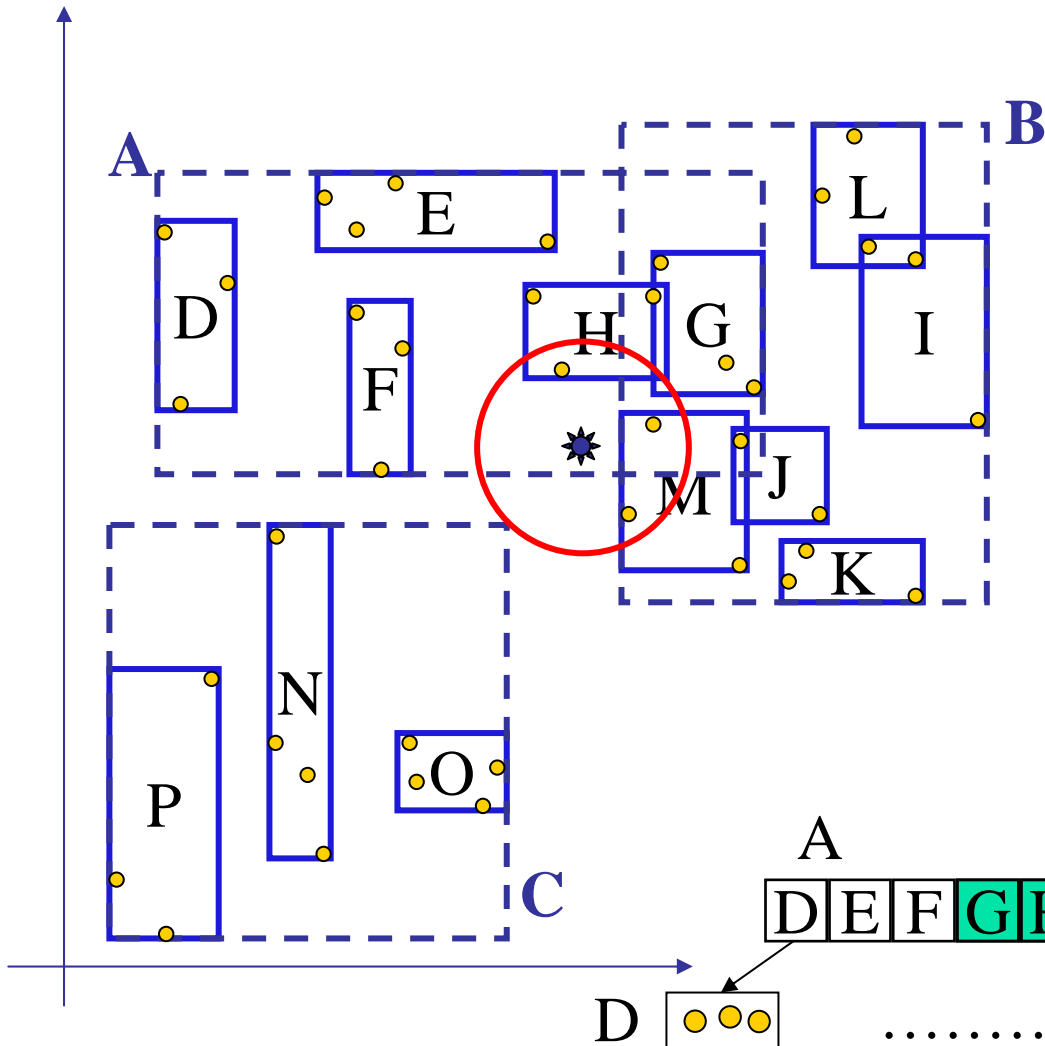
# Indici multi-dimensionali (spaziali)

---

- Per indicizzare efficacemente oggetti spaziali (poligoni, linee, punti,...) è possibile fare uso di indici appositi che lavorano su più coordinate
- Tra questi, senz'altro il più diffuso è l'**R-tree** (Guttman, 84), che riprende ed estende le idee base del B<sup>+</sup>-tree
- R-tree organizza i dati mediante (iper-)rettangoli, più precisamente **MBR** (**Minimum Bounding Rectangle**)
- Se i dati sono:
  - **Punti**: l'MBR degenera nel punto stesso
  - **Linee orizzontali o verticali**: l'MBR degenera nella linea stessa
  - **Poligoni o linee oblique**: l'MBR è **il più piccolo rettangolo (con i lati paralleli agli assi) che include l'oggetto**



# R-tree: l'intuizione



- Gli MBR vengono aggregati ricorsivamente
- Ne risulta un albero bilanciato, in cui gli MBR possono sovrapporsi

- Ogni elemento di un nodo non-foglia contiene:
  - Un puntatore al nodo figlio
  - La descrizione del suo MBR

# Indici metrici (distance-based)

---

- In molti casi si vogliono trovare oggetti che si trovano a una certa **distanza da un oggetto di riferimento** (query object), ma non necessariamente si hanno delle coordinate con cui ragionare
  - Es.: **edit distance**: minimo numero di caratteri che bisogna inserire, cancellare o sostituire per trasformare una stringa in un'altra

$d_{\text{edit}}(\text{'ball'}, \text{'bull'}) = 1$

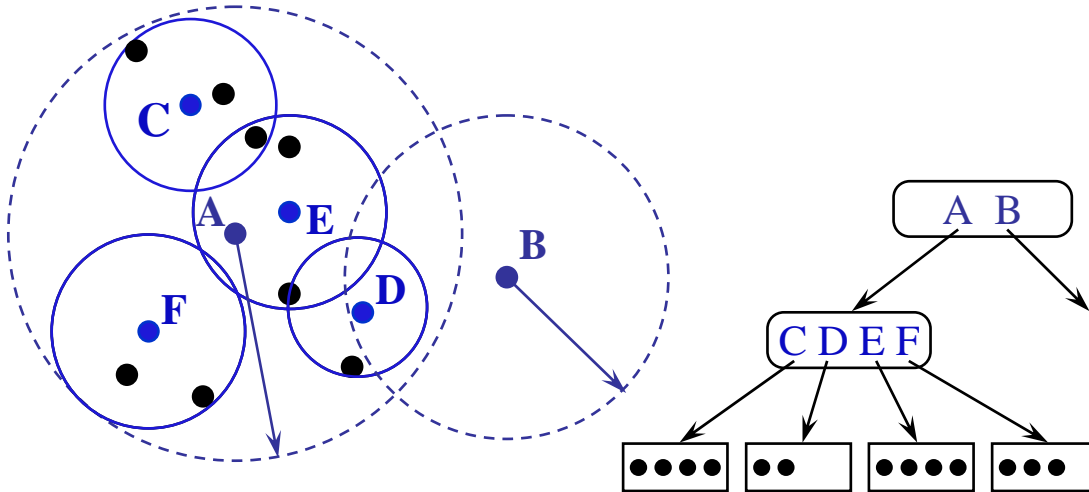
$d_{\text{edit}}(\text{'balls'}, \text{'bell'}) = 2$

$d_{\text{edit}}(\text{'rather'}, \text{'alter'}) = 3$

- Si parla in questo caso di indici metrici, tra cui si ritrova l'**M-tree** (Ciaccia, Patella, Zezula, 97)
- Sorgenti in C++ basati su GiST: <http://www-db.deis.unibo.it/Mtree/>
- M-tree si basa su principi simili a quelli dell'R-tree, e li estende al caso di metriche arbitrarie

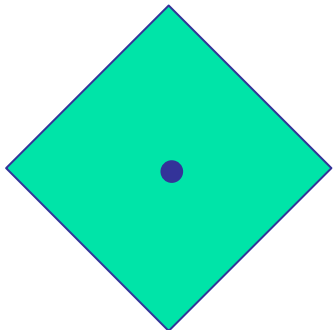
# M-tree: l'intuizione

distanza Euclidea

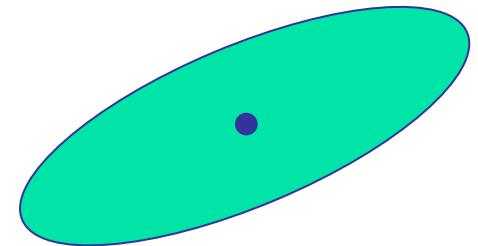
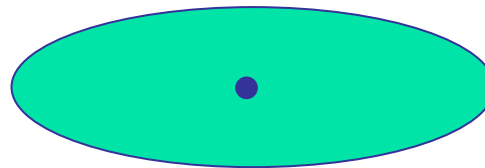
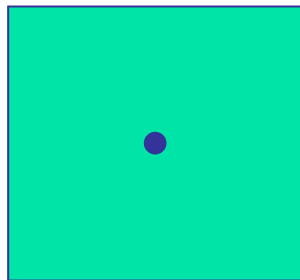


- Ogni elemento di un nodo non-foglia contiene:
  - Un puntatore al nodo figlio
  - La descrizione della regione:
    - “centro”
    - distanza max dal centro (“raggio”)

- La “forma” delle regioni dipende dalla metrica usata



Lezione 2



# Indici in PostgreSQL (1)

---

- In PostgreSQL si hanno diversi tipi di indici, B<sup>+</sup>-tree essendo il default

```
CREATE INDEX actor_idx ON FILMS (actor);
```

- Indici **UNIQUE** vengono creati automaticamente quando si dichiara tale vincolo o quello di **PRIMARY KEY** su una table, ma si possono anche creare esplicitamente

```
CREATE UNIQUE INDEX title_idx ON FILMS (title);
```

- Indici **HASH** richiedono più tempo per essere creati, e l'uso ne viene sconsigliato

```
CREATE INDEX cf_idx ON PERSONS USING hash (cf);
```

- Indici **parziali** servono, ad esempio, a escludere valori comuni o valori di scarso interesse per le query

```
CREATE INDEX enrolled_idx ON STUDENTS (enrolled)  
WHERE enrolled >= DATE '1980-01-01';
```



# Indici in PostgreSQL (2)

---

- Si possono definire anche indici **multi-attributo**

```
CREATE INDEX esami_idx ON Esami (CodCorso, Data, Voto);
```

- Va tenuto presente che indici di questo tipo sono utili solo se si specificano, in fase di query, **condizioni su tutti gli attributi o su un prefisso degli stessi** (es. su CodCorso e Data)
- Si possono anche definire indici **funzionali** (su espressioni)

```
CREATE INDEX aplusb_idx ON R (a+b);
```

# Indici in PostgreSQL (3): GiST

---

- **GiST** (Generalized Search Tree) è un **template** per creare tipi arbitrari di indici ad albero (B-tree, R-tree, M-tree, ...)  
<http://gist.cs.berkeley.edu/>  
<http://www.sai.msu.su/~megera/postgres/gist/>
- Lo **scopo** di GiST è quindi **semplificare lo sviluppo di nuovi tipi di indici**, garantendo al tempo stesso omogeneità e coerenza
- Per istanziare GiST è necessario, tra le altre cose, definire i seguenti metodi:
  - **Consistent**: determina quando una regione soddisfa la condizione della query
  - **Union**: calcola l'“unione” di un insieme di chiavi (es. MBR di più MBR)
  - **Penalty**: definisce una “penalità” quando si inserisce una chiave in una regione (es. aumento d'area dell'MBR, aumento del raggio)
  - **Picksplit**: usato per determinare come suddividere una regione
  - **Equal**: confronta 2 chiavi e ritorna true se sono uguali

# Indici in MySQL

---

- MySQL supporta:
  - indici B<sup>+</sup>-tree (default),
  - indici per dati spaziali (B<sup>+</sup>-tree o R-tree, questi ultimi solo per table MyISAM)
  - indici hash (per table MEMORY, ma non per table MyISAM e InnoDB)
- In MySQL la creazione degli indici può anche avvenire contestualmente alla creazione della table

```
CREATE TABLE lookup (id INT, INDEX id_idx (id));
```

# Indici: altri usi importanti

---

- Oltre che per restringere l'insieme dei dati cui accedere, un indice può servire a reperire i record secondo un certo ordine, e quindi può anche essere utile se nella query ci sono clausole ORDER BY e GROUP BY

```
SELECT *  
FROM Studenti  
ORDER BY datanascita
```

- Per alcune query la presenza di un indice può inoltre evitare di dover accedere alla table!

```
SELECT COUNT(DISTINCT datanascita)  
FROM Studenti
```

Se esiste un indice su (CodDip, Stipendio), anche la seguente query si può risolvere a livello di indice:

```
SELECT CodDip, MAX(Stipendio)  
FROM Impiegati  
GROUP BY CodDip
```

# Indici e aggiornamenti

---

- Poiché ogni indice deve riflettere la situazione corrente nel file dati, è evidente che **ogni operazione di modifica dei dati si deve propagare anche agli indici interessati**. Ad esempio:

```
UPDATE   Employee
SET      Salary = 2 * Salary
WHERE    Job = 'SQL Programmer'
```

richiede che si modifichino anche tutti gli indici costruiti su Salary

- Pertanto, se un indice viene poco usato in fase di interrogazione, non solo serve a poco costruirlo, ma la sua presenza può dar luogo a un **peggioramento di prestazioni**

# Il DB fisico di PostgreSQL: tablespaces

---

- PostgreSQL organizza i DB gestiti da un server (DB cluster) in **tablespace**, ognuno corrispondente a una directory (inizialmente vuota) del file system
- Usi tipici: **controllare lo spazio occupato dal DB** e **migliorare le prestazioni** (allocando dati frequentemente usati su dischi veloci e affidabili)

```
CREATE TABLESPACE fastspace LOCATION '/mnt/sda1/pgsql/data';
```

```
CREATE TABLE foo(i int) TABLESPACE fastspace;
```

ovvero:

```
SET default_tablespace = fastspace;
```

```
CREATE TABLE foo(i int);
```

- Elenco dei tablespace: `SELECT spcname FROM pg_tablespace;`
- Definizione del tablespace di default per un DB:

```
CREATE DATABASE new_db TABLESPACE fastspace;
```

# Il DB fisico di PostgreSQL: partitioning (1)

---

- Tipicamente una table SQL è memorizzata in un singolo file
- Per motivi essenzialmente prestazionali, è possibile partizionare la table, secondo una delle seguenti modalità:
  - **Range partitioning**: ogni partizione contiene un certo range di chiavi
  - **List partitioning**: vengono elencate esplicitamente le chiavi per ogni partizione
- In entrambi i casi, il tutto si basa su una caratteristica *object-relational* di PostgreSQL, ovvero la **table inheritance**

# PostgreSQL: table inheritance (sketch)

---

- Tale caratteristica risponde a necessità di modellazione che non sono naturalmente soddisfatte dal modello relazionale
- Ad es., data la “parent” table

```
CREATE TABLE cities (name text,  
                      population float,  
                      altitude int);
```

si può definire una “child” table che eredita gli attributi di `cities`, oltre a vincoli `CHECK` e `NOT NULL` (ma non altri, quali `PRIMARY KEY`)

```
CREATE TABLE capitals (state char(2))  
INHERITS (cities);
```

- Per vedere tutte le città: `SELECT * FROM cities`
- Per vedere solo le città non capitali: `SELECT * FROM ONLY cities`
- Maggiori dettagli: cap. 5.8



# Il DB fisico di PostgreSQL: partitioning (2)

---

- Per realizzare un range partitioning:
  - Si definisce una “**master**” table (senza dati)
  - Si definiscono tante “**child**” tables quante necessarie, con vincoli opportuni di range

```
CREATE TABLE students ( stud_id int not null,  
                        enrolled date not null, ... );
```

```
CREATE TABLE old_students (  
    CHECK (enrolled < DATE '2000-01-01'))  
    INHERITS(students) TABLESPACE archival;
```

```
CREATE TABLE new_students (  
    CHECK (enrolled >= DATE '2000-01-01'))  
    INHERITS(students);
```

# Il DB fisico di PostgreSQL: partitioning (3)

---

- Uno dei vantaggi di avere table partizionate è quello di poter restringere l'insieme dei dati da esaminare per risolvere una query

- Ad esempio:

```
SET constraint_exclusion = on;  
SELECT count(*)  
FROM students  
WHERE enrolled >= DATE '2006-01-01';
```

- Attivando `constraint_exclusion`, il query processor di PostgreSQL verifica, esaminando le relative clausole `CHECK`, quali partizioni di `students` possono contribuire al risultato (nel nostro esempio solo `new_students`)
- Ne consegue un notevole risparmio di risorse

# Il DB fisico di MySQL: storage engines

---

- MySQL mette a disposizione diversi tipi di **storage engine**, che forniscono caratteristiche e prestazioni differenziate. Tra questi:
  - **MyISAM**: gestisce table **non transazionali(\*)**. Buone prestazioni e caratteristiche di full-text search. E' lo storage engine di default
  - **InnoDB**: supporta transazioni ACID e offre ripristino da guasti (forum: <http://forums.mysql.com/list.php?22>)
  - **MEMORY**: table i cui dati risiedono in memoria (lo schema è però salvato su disco). Di default usano indici hash, e sono tipicamente usate per creare table temporanee

```
CREATE TABLE FOO(i int) ENGINE = InnoDB;
```

- Dettagli: cap. 14

(\*) In una table non transazionale non è possibile fare rollback e ripristinare da guasti. Si veda più avanti la parte di gestione delle transazioni.

# Il DB fisico di MySQL: MyISAM

---

- Ogni table MyISAM è memorizzata in 3 file (dati, formato e indici)
- Per ragioni prestazionali è possibile creare file dati e indici in directory differenti
  - `DATA DIRECTORY = 'absolute path to directory'`
  - `INDEX DIRECTORY = 'absolute path to directory'`
- Il formato di memorizzazione dei dati dipende dai campi, e può essere
  - statico (record a lunghezza fissa),
  - dinamico (record a lunghezza variabile),
  - compresso (solo table read-only, `myisampack` tool)
- Una table MyISAM supporta **inserimenti in concorrenza a letture** (query) **solo se non ci sono “buchi”**, dovuti a cancellazioni, nel file
  - In questo caso gli inserimenti avvengono in coda al file e non interferiscono con le letture

# Il DB fisico di MySQL: InnoDB

---

- Lo storage engine InnoDB organizza i dati in tablespaces, definiti in fase di configurazione (`my.cnf`):

```
innodb_data_home_dir = /ibdata
```

```
innodb_data_file_path = ibdata1:50M;ibdata2:50M:autoextend  
[ :max:500M]
```

- Se lo spazio a disposizione si esaurisce, è necessario arrestare il server e aggiungere un altro tablespace (solo l'ultimo può essere **autoextendable**)
- Ogni table InnoDB ha un **clustered index** che contiene i dati.
  - Tale indice è costruito sulla **PRIMARY KEY** (se definita),
  - altrimenti sul primo campo **UNIQUE NOT NULL**,
  - altrimenti sul "row ID" progressivo che InnoDB assegna a ogni record
- Gli altri indici (detti "secondary") contengono il valore usato dal clustered index (**puntatore logico** anziché fisico)

# Gestione delle transazioni

---

# Cos'è una transazione?

---

- Una **transazione** è un'unità logica di elaborazione che corrisponde a una serie di operazioni fisiche elementari (letture/scritture) sul DB

- Esempi:

- Trasferimento di una somma da un conto corrente ad un altro

```
UPDATE CC
```

```
SET Saldo = Saldo - 50
```

```
WHERE Conto = 123
```

```
UPDATE CC
```

```
SET Saldo = Saldo + 50
```

```
WHERE Conto = 235
```

- Aggiornamento degli stipendi degli impiegati di una sede

```
UPDATE Imp
```

```
SET Stipendio = 1.1*Stipendio
```

```
WHERE Sede = 'S01'
```

- In entrambi i casi **tutte le operazioni elementari devono essere eseguite**

# Proprietà ACID di una transazione

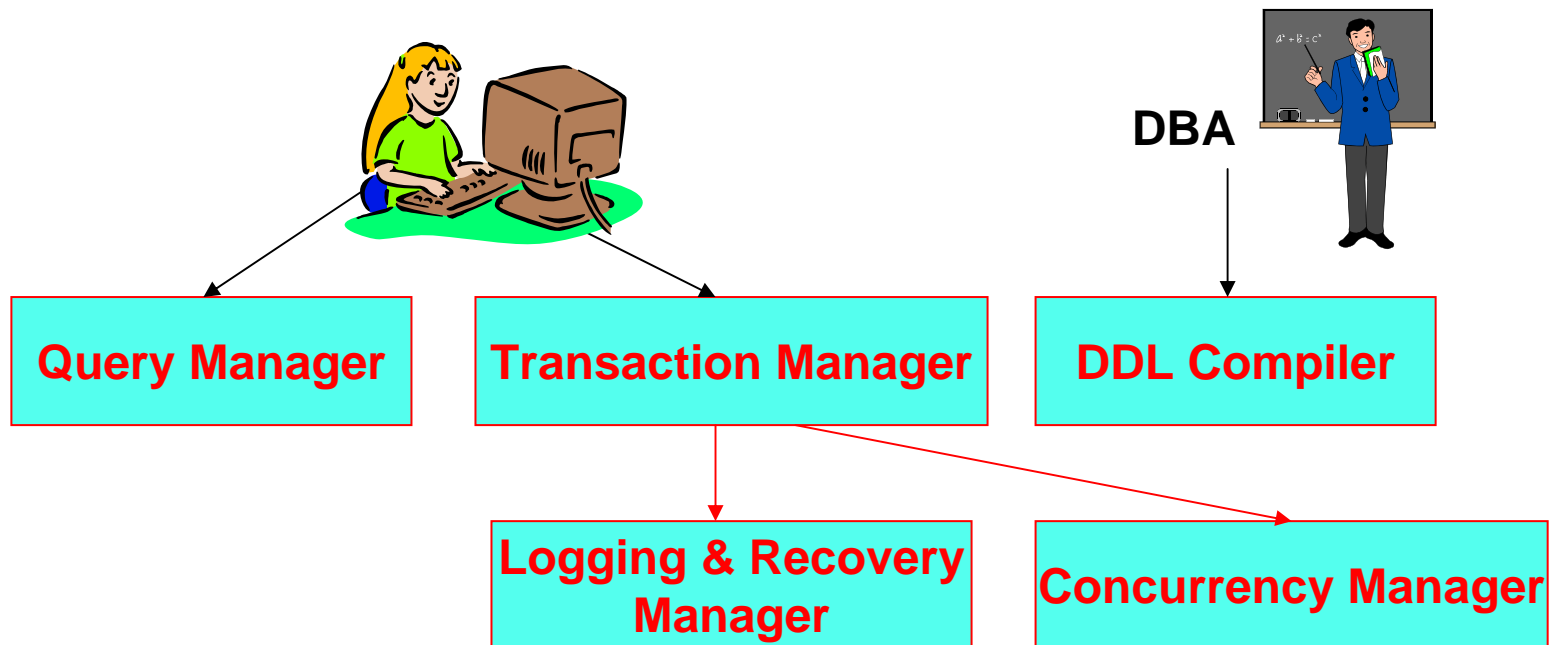
---

- L'acronimo **ACID** indica le 4 proprietà che il DBMS deve garantire che valgano per ogni transazione:
  - **Atomicity** = una transazione è un'unità di elaborazione
    - Il DBMS garantisce che la transazione venga eseguita come un tutt'uno
  - **Consistency** = una transazione lascia il DB in uno stato consistente
    - Il DBMS garantisce che nessuno dei vincoli di integrità del DB venga violato
  - **Isolation** = una transazione esegue indipendentemente dalle altre
    - Se più transazioni eseguono in concorrenza, il DBMS garantisce che l'effetto netto è equivalente a quello di una qualche esecuzione sequenziale delle stesse
  - **Durability** = gli effetti di una transazione che ha terminato correttamente la sua esecuzione devono essere persistenti nel tempo
    - Il DBMS deve proteggere il DB a fronte di guasti



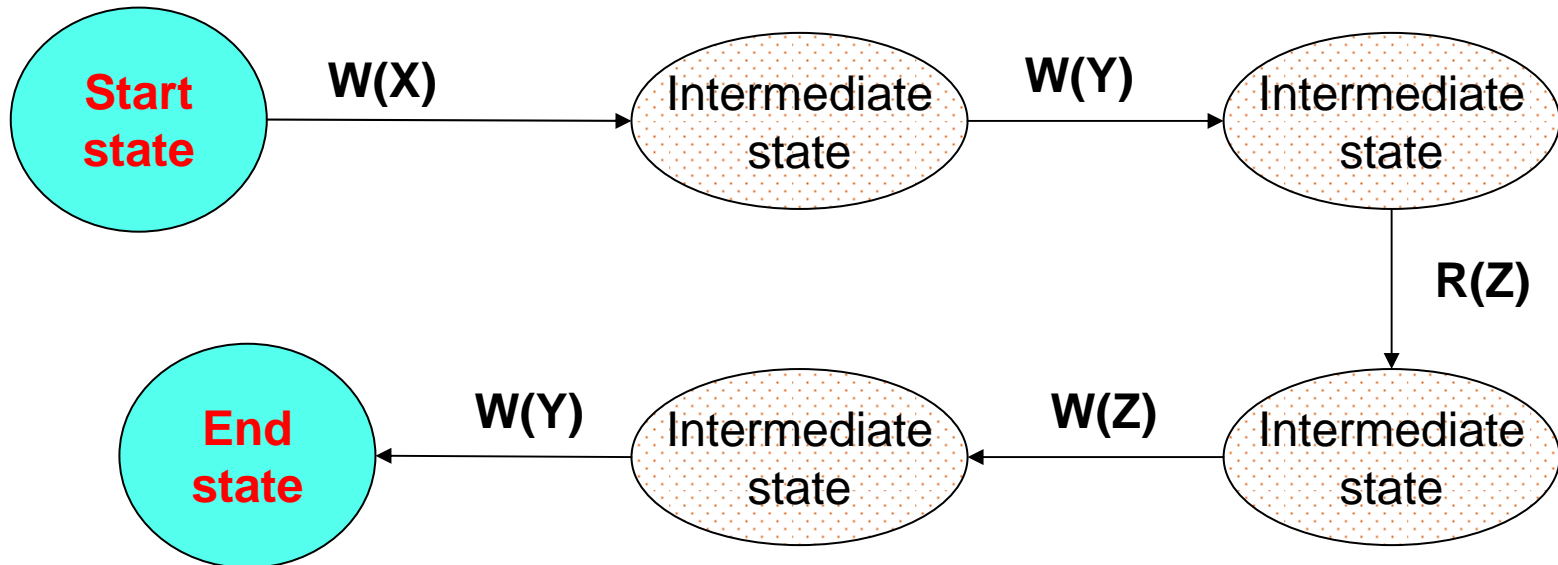
# Proprietà ACID e moduli di un DBMS

<b>Transaction Manager :</b>	coordina l'esecuzione delle transazioni, ricevendo i comandi SQL ad esse relativi
<b>Logging &amp; Recovery Manager:</b>	si fa carico di <b>Atomicity</b> e <b>Durability</b>
<b>Concurrency Manager:</b>	garantisce l' <b>Isolation</b>
<b>DDL Compiler:</b>	genera i controlli per la <b>Consistency</b>



# Modello delle transazioni

- Nel modello che consideriamo una transazione viene vista come una sequenza di operazioni elementari di lettura (R) e scrittura (W) di oggetti (tuple) del DB che, a partire da uno stato iniziale consistente del DB, porta il DB in un nuovo stato finale consistente



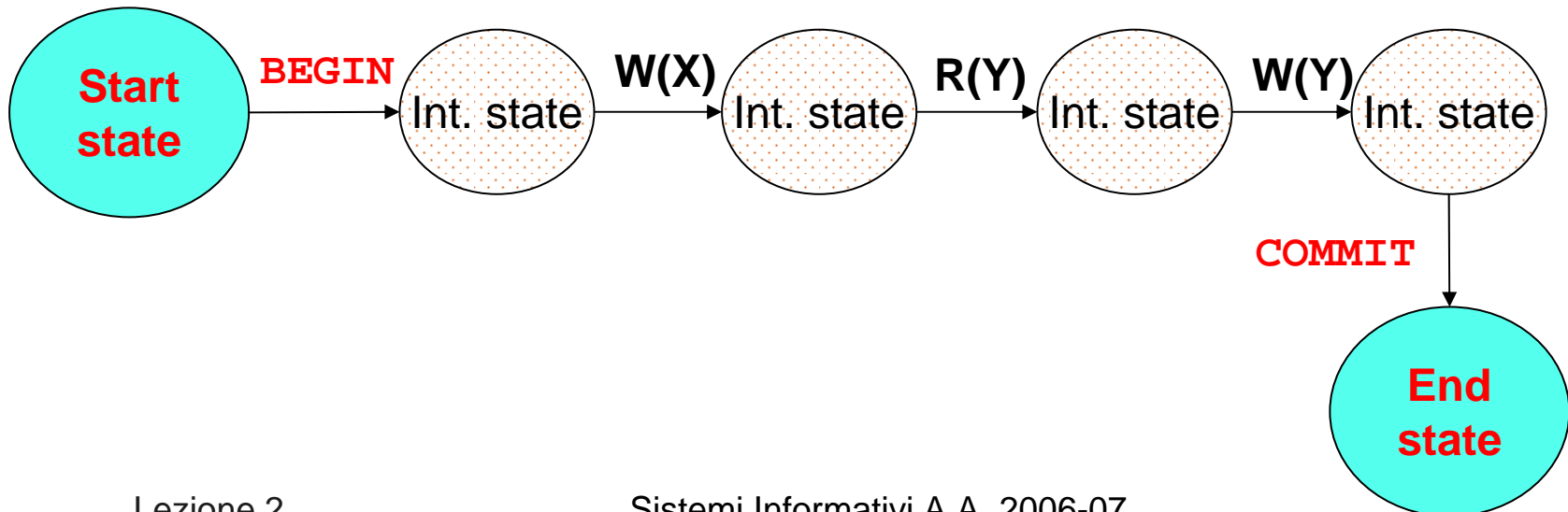
- In generale gli stati intermedi in cui si trova il DB non è richiesto che siano consistenti

# Possibili esiti di una transazione (1)

- Nel modello considerato una transazione (il cui inizio viene indicato dalla parola chiave **BEGIN [WORK]**, anche se in SQL standard è implicito) può avere solo 2 esiti:

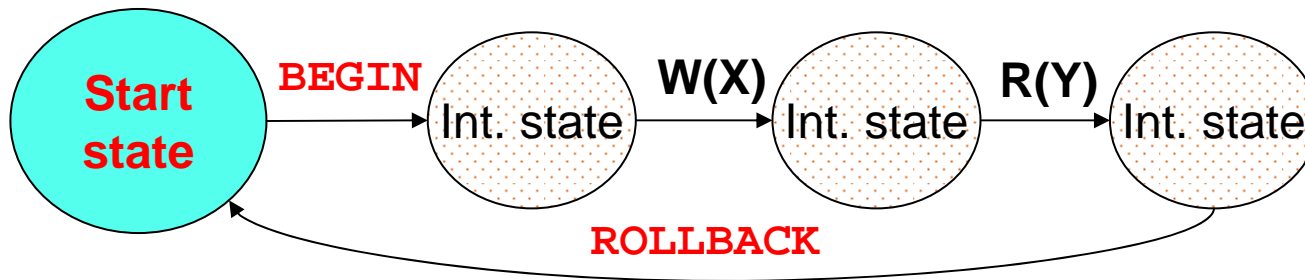
- Terminare **correttamente**:

Questo avviene solo quando l'applicazione, dopo aver eseguito tutte le proprie operazioni, esegue una particolare istruzione SQL, detta **COMMIT** (o **COMMIT WORK**), che comunica "ufficialmente" al Transaction Manager il termine delle operazioni



# Possibili esiti di una transazione (2)

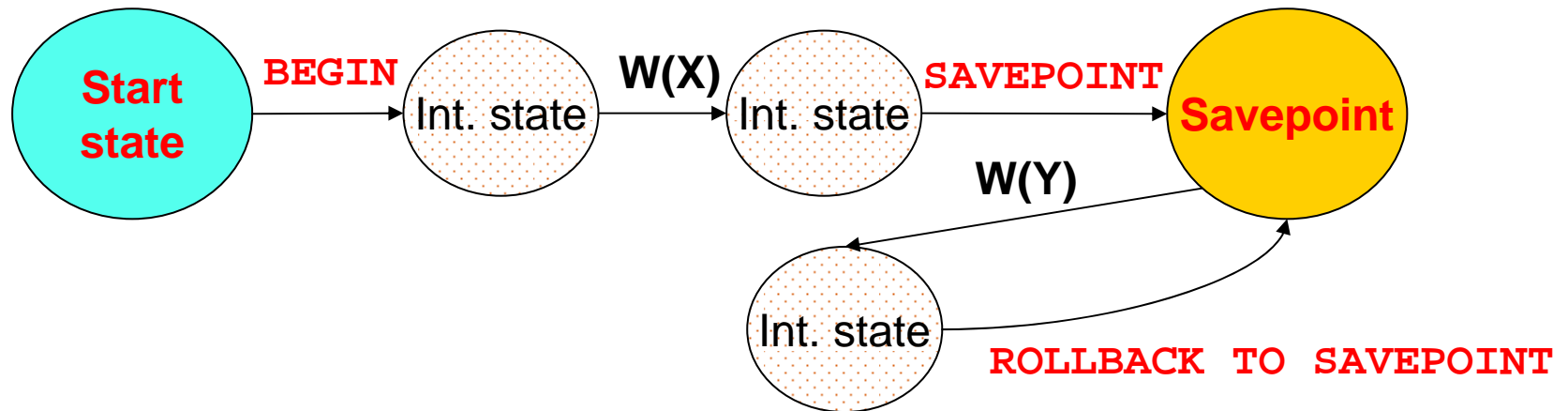
- Terminare **non correttamente** (anticipatamente); sono possibili 2 casi:
  - È la transazione che, per qualche motivo, decide che non ha senso continuare e quindi “abortisce” eseguendo l’istruzione SQL **ROLLBACK** (o **ROLLBACK WORK**)
  - È il sistema che non è in grado (ad es. per un guasto o per la violazione di un vincolo) di garantire la corretta prosecuzione della transazione, che viene quindi abortita



- Se per qualche motivo la transazione non può terminare correttamente la sua esecuzione il DBMS deve “disfare” (**UNDO**) le eventuali modifiche da essa apportate al DB

# Transazioni con Savepoint

- Il modello di transazioni usato dai DBMS è in realtà più articolato; in particolare è possibile definire dei cosiddetti “**savepoint**”, che vengono utilizzati da una transazione per **disfare solo parzialmente il lavoro svolto**



# Esempio di transazione con Savepoint

---

**BEGIN WORK**

**SELECT \* FROM Department**

**INSERT INTO Department (DeptNo, DeptName, AdmrDept)  
VALUES ('X00', 'nuovo dept 1', 'A00')**

**SAVEPOINT pippo**

**SELECT \* FROM Department -- qui include 'X00'**

**INSERT INTO Department (DeptNo, DeptName, AdmrDept)  
VALUES ('Y00', 'nuovo dept 2', 'A00')**

**SELECT \* FROM Department -- qui anche 'Y00'**

**ROLLBACK WORK TO SAVEPOINT pippo**

**SELECT \* FROM Department -- qui solo 'X00'**

**COMMIT WORK**

# Esecuzione seriale e concorrente

---

- Un DBMS, dovendo supportare l'esecuzione di diverse transazioni che accedono a dati condivisi, potrebbe eseguire tali transazioni in sequenza (“**serial execution**”)
- In alternativa, il DBMS può eseguire più transazioni in concorrenza, alternando l'esecuzione di operazioni di una transazione con quella di operazioni di altre transazioni (“**interleaved execution**”)
- Eseguire più transazioni concorrentemente è necessario per garantire **buone prestazioni**:
  - Si sfrutta il fatto che, mentre una transazione è in attesa del completamento di una operazione di I/O, un'altra può utilizzare la CPU, il che porta ad aumentare il “**throughput**” (**n. transazioni elaborate nell'unità di tempo**) del sistema
  - Se si ha una transazione “breve” e una “lunga”, **l'esecuzione concorrente porta a ridurre il tempo medio di risposta del sistema**

# Isolation: gestire la concorrenza

---

- Quando più transazioni vengono eseguite in concorrenza, si ha “isolation” se non vi è interferenza tra le stesse, ovvero **se gli effetti sul DB sono equivalenti a quelli che si avrebbero con una esecuzione serializzata**
- Nel caso di transazioni che interferiscono tra loro si possono avere 4 tipi base di problemi (dal più grave al meno grave):
  - **Lost Update:** un aggiornamento viene perso
  - **Dirty Read:** viene letto un dato che “non esiste” nel DB
  - **Unrepeatable Read:** letture successive di uno stesso dato sono tra loro inconsistenti
  - **Phantom Row:** vengono selezionati dei dati di interesse, ma tra questi ne manca qualcuno (*phantom*)



# Lost Update

- Il seguente schedule mostra un caso tipico di lost update, in cui per comodità si evidenziano anche le operazioni che modificano il valore del dato X e si mostra come varia il valore di X nel DB

**Questo update viene perso!**

T1	X	T2
R(X)	1	
X=X-1	1	
	1	R(X)
	1	X=X-1
W(X)	0	
Commit	0	
	0	W(X)
	0	Commit

- Problema: T2 legge il valore di X prima che T1 (che lo ha già letto) lo modifichi

# Dirty Read

- In questo caso il problema è che una transazione legge un dato “che non c'è”:

T1	X	T2
R(X)	0	
X=X+1	0	
W(X)	1	
	1	R(X)
Rollback	0	
	0	...
	0	Commit

Questa lettura è “sporca”!

- Quanto svolto da T2 si basa su un valore di X “intermedio”, e quindi non stabile
- Le conseguenze sono imprevedibili (dipende cosa fa T2) e si presenterebbero anche se T1 non abortisse

# Unrepeatable Read

- Ora il problema è che una transazione legge due volte un dato e trova valori diversi:

T1	X	T2
R(X)	0	
	0	R(X)
	1	X=X+1
	1	W(X)
	1	Commit
R(X)	1	
Commit	1	

**Le 2 letture sono tra loro inconsistenti!**

- Anche in questo caso si possono avere gravi conseguenze
- Lo stesso problema si presenta per **transazioni di “analisi”**
  - Ad esempio T1 somma l'importo di 2 conti correnti mentre T2 esegue un trasferimento di fondi dall'uno all'altro (T1 potrebbe quindi riportare un totale errato)

# Phantom Row

- Questo caso si può presentare quando vengono **inserite o cancellate tuple che un'altra transazione dovrebbe logicamente considerare**
  - Nell'esempio la tupla t4 è un **“phantom”**, in quanto T1 “non la vede”

**T1:**

```
UPDATE Prog
SET Sede = 'Firenze'
WHERE Sede = 'Bologna'
```

**T2:**

```
INSERT INTO Prog
VALUES ('P03', 'Bologna')
```

**Prog**

CodProg	Citta	
P01	Milano	t1
P01	Bologna	t2
P02	Bologna	t3
<b>P03</b>	<b>Bologna</b>	<b>t4</b>

**T1 “non vede”  
questa tupla!**

T1	T2
R(t2)	
R(t3)	
...	
W(t2)	
W(t3)	
	<b>Insert(t4)</b>
...	
Commit	
	<b>Commit</b>

# Come garantire l'Isolation

- Una comune tecnica usata dai DBMS per evitare i problemi visti consiste nell'uso di **lock**
  - Per eseguire un'operazione è prima necessario “acquisire” un lock sulla risorsa interessata (ad es. una tupla)
- I lock sono di vario tipo; quelli di base sono:
  - **S** (Shared): un lock condiviso è necessario per leggere
  - **X** (eXclusive): un lock esclusivo è necessario per scrivere/modificare
- Quando una transazione T vuole operare su un dato Y, viene inviata la richiesta di acquisizione del lock corrispondente, e il lock viene accordato a T in funzione della seguente **tabella di compatibilità**
- Quando T ha terminato di usare Y, può **rilasciare** il lock (**unlock(Y)**)

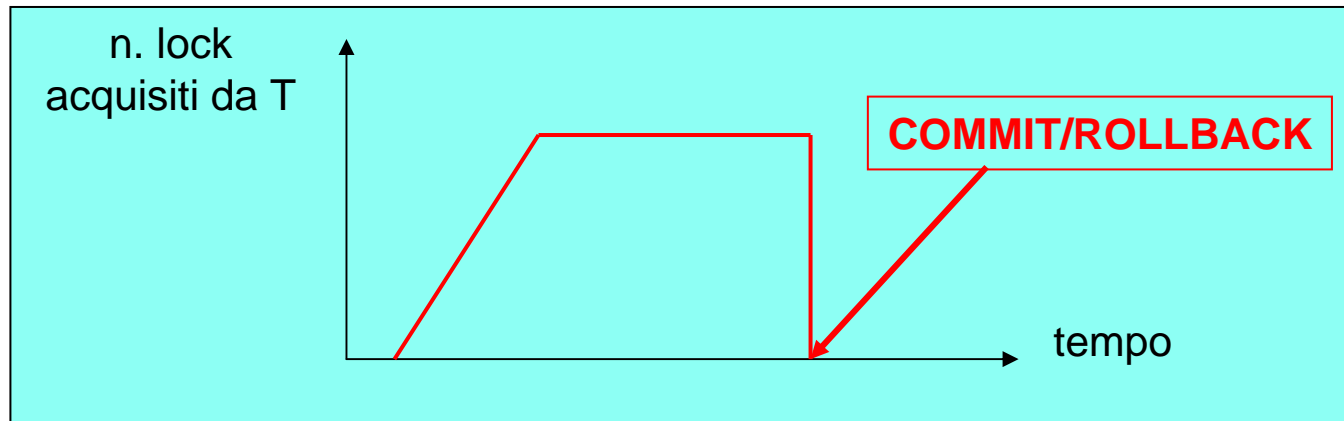
Su Y un'altra transazione ha un lock di tipo

T richiede un lock di tipo

	S	X
S	OK	NO
X	NO	NO

# Protocollo Strict 2-phase lock (Strict 2PL)

- Il modo con cui le transazioni rilasciano i lock acquisiti è la chiave per risolvere i problemi di concorrenza
- Si può dimostrare che se
  - Una transazione prima acquisisce tutti i lock necessari
  - Rilascia i lock solo al termine dell'esecuzione (**COMMIT** o **ROLLBACK**)allora l'Isolation è garantita



- Come effetto collaterale si possono verificare **deadlock**, ossia situazioni di stallo, che vengono risolte facendo abortire una transazione

# Lock: assenza di Dirty Read

- A titolo di esempio, si fa vedere come l'uso dei lock permette di risolvere il problema delle dirty reads

T1	X	T2
<b>S-lock(X)</b>	<b>0</b>	
R(X)	0	
X=X+1	0	
<b>X-lock(X)</b>	<b>0</b>	
W(X)	1	
	1	<b>S-lock(X)</b>
	1	wait
Rollback	0	wait
<b>Unlock(X)</b>	<b>0</b>	wait
	<b>0</b>	R(X)

- In questo caso T2 deve aspettare la terminazione di T1 prima di poter leggere il valore di X

# Livelli di isolamento in SQL

---

- Scegliere di operare a un livello di isolamento in cui si possono presentare dei problemi ha il vantaggio di aumentare il grado di concorrenza raggiungibile, e quindi di migliorare le prestazioni
- Lo standard SQL definisce 4 livelli di isolamento (YES significa che il problema può presentarsi):

Isolation Level	Lost Update	Dirty Read	Unrepeatable Read	Phantom
<b>SERIALIZABLE</b>	NO	NO	NO	NO
<b>REPEATABLE READ</b>	NO	NO	NO	<b>YES</b>
<b>READ COMMITTED</b>	NO	NO	<b>YES</b>	<b>YES</b>
<b>READ UNCOMMITTED</b>	NO	<b>YES</b>	<b>YES</b>	<b>YES</b>



# Concorrenza in MySQL e PostgreSQL

---

- In **MySQL** il livello di default è **REPEATABLE READ**; per cambiarlo (prima di una transazione), ad es. a **SERIALIZABLE**, si usa l'istruzione SQL

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
```

- In **PostgreSQL** è possibile richiedere ognuno dei 4 livelli (il default è **READ COMMITTED**), ma ne vengono supportati solo 2, come da tabella:

Richiesto	Effettivo
SERIALIZABLE	SERIALIZABLE
REPEATABLE READ	SERIALIZABLE
READ COMMITTED	READ COMMITTED
READ UNCOMMITTED	READ COMMITTED

- Il motivo è dovuto alla modalità con cui PostgreSQL gestisce transazioni in concorrenza, noto come **Multiversion Concurrency Control (MVCC)**

# Atomicity e Durability: convivere con i guasti

---

- L'“altra faccia” della gestione delle transazioni riguarda il trattamento dei guasti (“**failure**”), ovvero di tutti quegli **eventi anomali che possono pregiudicare il corretto funzionamento delle transazioni**
- I tipi di malfunzionamenti sono essenzialmente 3:
  - **Transaction failure**: è il caso in cui una transazione abortisce
  - **System failure**: il sistema ha un guasto hardware o software che provoca l'interruzione di tutte le transazioni in esecuzione, senza però danneggiare la memoria permanente (dischi)
  - **Media (o device) failure**: in questo caso il contenuto (persistente) della base di dati viene danneggiato

# Atomicity e Durability: il Log

---

- Per far fronte ai malfunzionamenti, un DBMS fa uso di diversi strumenti, in particolare:
  - **DataBase Dump**: copia di archivio del DB (o parte di esso)
  - **Log file** (“giornale”): file sequenziale in cui vengono registrate le operazioni di modifica eseguite dalle transazioni
- Se una pagina **P** del DB viene modificata dalla transazione **T**, il Log contiene un record del tipo  
**(LSN, T, PID, before(P), after(P), prevLSN)**
  - **LSN** = Log Sequence Number (n. progressivo del record nel Log)
  - **T** = identificatore della transazione
  - **PID** = identificatore della pagina modificata
  - **before(P)** = è la cosiddetta “**before image**” di P, ovvero il contenuto di P prima della modifica
  - **after(P)** = è l’“**after image**” di P, ossia il contenuto di P dopo la modifica
  - **prevLSN** = LSN del precedente record del Log relativo a T

# Esempio di Log

- Il Log contiene anche record che specificano l'inizio (**BEGIN**) di una transazione e la sua terminazione (**COMMIT** o **ROLLBACK**)

LSN	T	PID	before(P)	after(P)	prevLSN
...					
235	T1	BEGIN			-
236	T2	BEGIN			-
237	T1	P15	(abc, 10)	(abc, 20)	235
238	T2	P18	(def, 13)	(ghf, 13)	236
239	T1	COMMIT			237
240	T2	P19	(def, 15)	(ghf, 15)	238
241	T3	BEGIN			-
242	T2	P19	(ghf, 15)	(ghf, 17)	240
243	T3	P15	(abc, 20)	(abc, 30)	241
244	T2	ROLLBACK			242
245	T3	COMMIT			243
...					

# Protocollo WAL

---

- Affinché il Log possa essere utilizzato per ripristinare lo stato del DB a fronte di malfunzionamenti, è importante che venga applicato il cosiddetto protocollo **WAL** (“**Write-ahead Logging**”):

prima di scrivere su disco una pagina P modificata,  
il corrispondente Log record deve essere già stato scritto nel Log

- Intuitivamente, se il protocollo WAL non viene rispettato è possibile che
  - Una transazione T modifichi il DB aggiornando una pagina P
  - Si verifichi un system failure prima che il Log record relativo alla modifica di P sia stato scritto nel Log
- In questo caso non c'è alcun modo di riportare il DB allo stato iniziale
- Per vedere come il protocollo WAL può essere applicato è necessario parlare prima di **buffering delle pagine**

# Letture e scrittura via buffer

---

- La lettura di una tupla (o record) richiede che la pagina corrispondente sia prima portata in memoria, in un'area gestita dal DBMS detta **buffer pool**
  - Tipicamente, ogni tablespace ha un proprio buffer pool
- Ogni buffer nel pool può ospitare una copia di una pagina su disco
- **La gestione del buffer pool**, che è fondamentale sia dal punto di vista prestazionale che da quello della correttezza delle transazioni, è demandata a un modulo detto **Buffer Manager (BM)**
- Il BM è chiamato in causa anche nel caso di scritture, ovvero quando bisogna riscrivere su disco una pagina modificata

# Il Buffer Manager

---

- A fronte di una **richiesta di una pagina**, il Buffer Manager (BM) opera come segue:
  - Se la pagina è già in un buffer, viene fornito al richiedente l'indirizzo del buffer
  - Se la pagina non è in memoria:
    - Il BM seleziona un buffer per la pagina richiesta. Se tale buffer è occupato da un'altra pagina, 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
- La **scrittura di una pagina modificata** avviene come segue:
  - Il richiedente modifica la pagina nel buffer e ne informa il BM
  - Il BM, in maniera **asincrona**, provvede a scrivere la pagina su disco

# Interfaccia del Buffer Manager

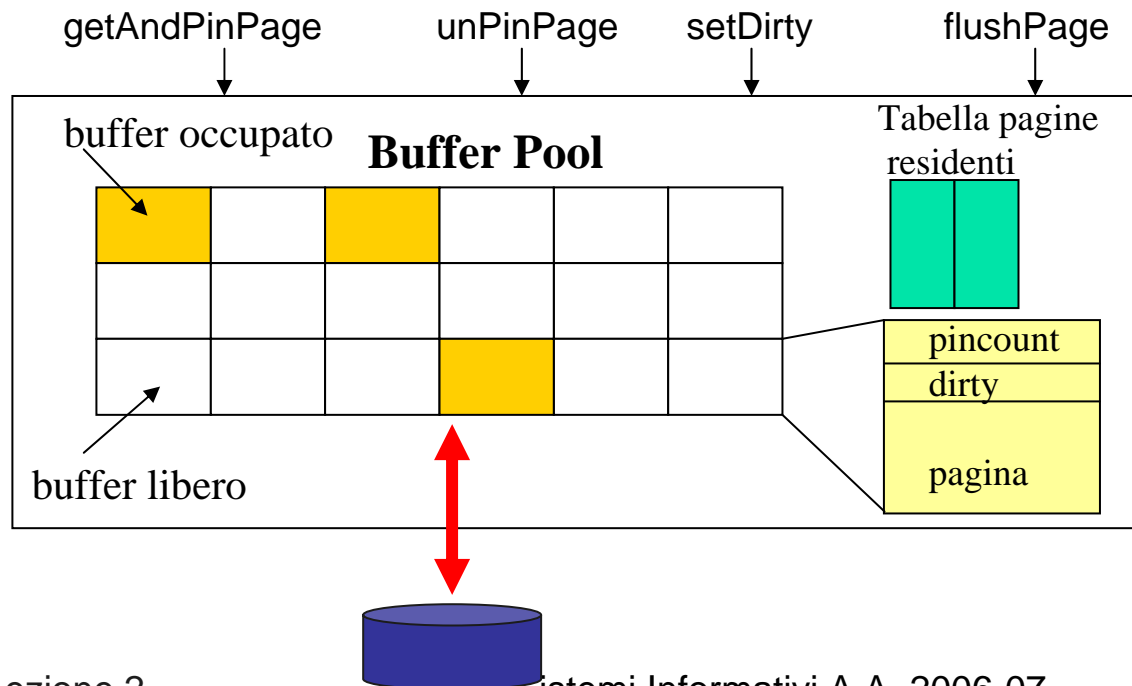
- L'interfaccia che il BM offre agli altri moduli del DBMS ha quattro metodi di base:

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

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

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

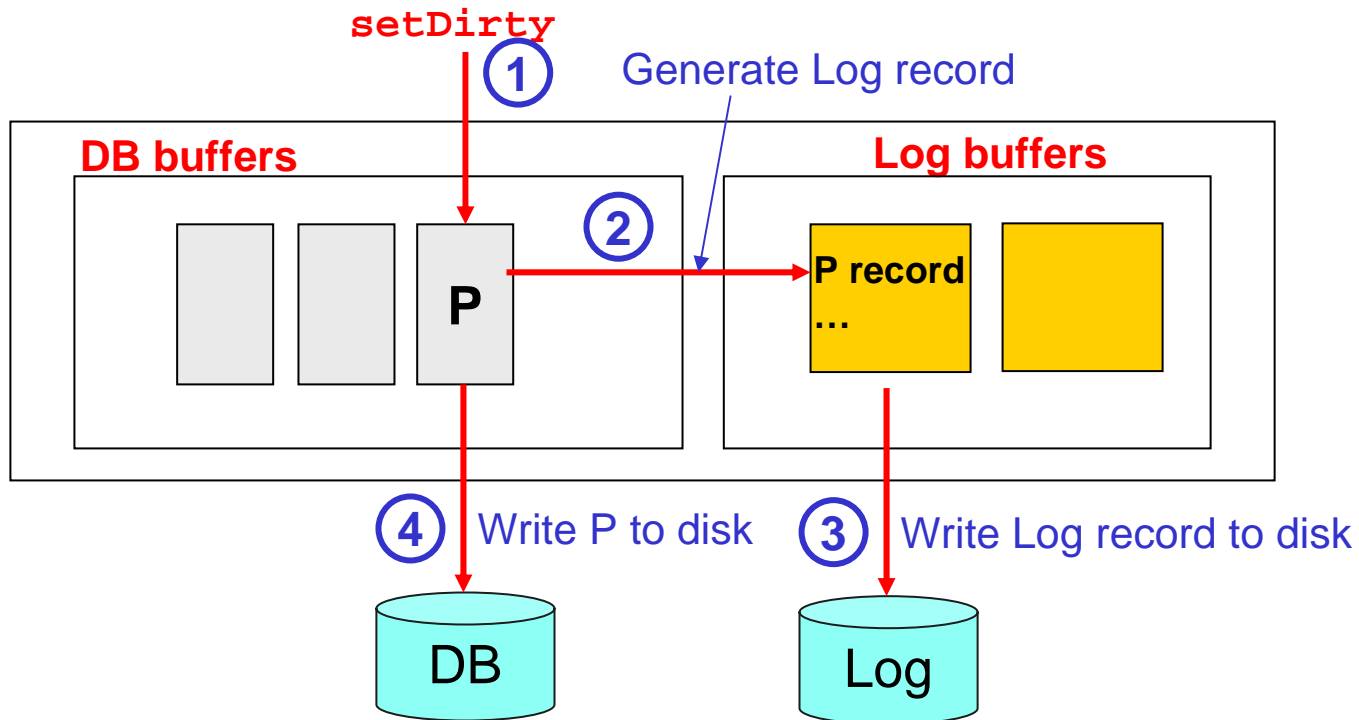
**flushPage**: forza la scrittura della pagina su disco, rendendola così “pulita”





# Implementazione del protocollo WAL

- La responsabilità di garantire il rispetto del protocollo WAL è del **Buffer Manager**, che gestisce, oltre ai buffer del DB, anche quelli del Log
- In figura viene riportato l'ordine in cui si succedono le varie operazioni relative alla modifica di una pagina P



# Gestione dei buffer

---

- Quando una transazione **T** modifica una pagina **P**, il Buffer Manager ha 2 possibilità:
  - Politica No-steal**: Mantenere la pagina **P** nel buffer, e attendere che **T** abbia eseguito **COMMIT** prima di scriverla su disco
  - Politica Steal**: Scrivere **P** quando “più conviene” (per liberare il buffer o per ottimizzare le prestazioni di I/O), eventualmente anche prima della terminazione di **T**
- E' evidente che nel primo caso non ci sarà mai bisogno di eseguire l’**“UNDO”** di transazioni che abortiscono, ma **si rischia di esaurire lo spazio a disposizione in memoria centrale** (in quanto una transazione non può “rubare” i buffer ad altre transazioni ancora in esecuzione)

# Esecuzione del Commit

---

- Quando una transazione esegue **COMMIT** si hanno ancora 2 possibilità:
  - Politica Force**: prima di scrivere il record di **COMMIT** sul Log, che “ufficializza” la conclusione della transazione, si forza la scrittura su disco di tutte le pagine modificate da T
  - Politica No-force**: si scrive subito il record di **COMMIT** sul Log; quindi, quando T termina, alcune delle sue modifiche potrebbero ancora non essere state rese persistenti su disco
- La politica **Force** può generare **molti I/O inutili**
  - Ad esempio, se una pagina P è frequentemente modificata (“**hot spot**”), deve essere scritta su disco ad ogni **COMMIT**
  - Con la politica No-force ciò non è necessario, e P viene scritta su disco solo se deve essere rimpiazzata nel buffer, secondo la politica adottata dal Buffer Manager

# Transaction failure

- Con la politica **Steal**, se una transazione T abortisce è possibile che alcune pagine da essa modificate siano già state scritte su disco
- Per annullare (**UNDO**) queste modifiche si scandisce il Log a ritroso (usando i prevLSN) e si ripristinano nel DB le **before image** delle pagine modificate da T

LSN	T	PID	before(P)	after(P)	prevLSN
...					
236	T2	BEGIN			-
237	T1	P15	(abc, 10)	(abc, 20)	235
238	T2	P18	(def, 13) ←	(ghf, 13)	236
239	T1	COMMIT			237
240	T2	P19	(def, 15) ←	(ghf, 15)	238
241	T3	BEGIN			-
242	T2	P19	(ghf, 15) ←	(ghf, 17)	240
243	T3	P15	(abc, 20)	(abc, 30)	241
244	T2	ROLLBACK			242

# System failure

- Nel caso di system failure, vengono disfatte tutte le transazioni il cui **COMMIT** record non si trova nel Log
- Se si adotta la politica **No-force** non è detto che tutte le modifiche operate da una transazione T che ha eseguito **COMMIT** siano state riportate su disco; pertanto T va “rifatta” (**REDO**), riscrivendo le **after image** che si trovano sul Log

LSN	T	PID	before(P)	after(P)	prevLSN
...					
235	T1	BEGIN			-
236	T2	BEGIN			-
237	T1	P15	(abc, 10)	→ (abc, 20)	235
238	T2	P18	(def, 13)	(ghf, 13)	236
239	T1	COMMIT			237
...					

# Checkpoint

- La procedura di **restart** è quella che si occupa di riportare il DB in uno stato consistente a fronte di system failure; per ridurre i tempi di restart, periodicamente si può eseguire un “**checkpoint**”, ovvero una **scrittura forzata su disco delle pagine modificate**
  - L’esecuzione del checkpoint viene registrata scrivendo sul Log un record **CKP** (checkpoint)
- In questo modo se T ha eseguito **COMMIT** prima del checkpoint si è sicuri che T non deve essere rifatta

**T1 e T2 non devono essere rifatte!**

LSN	T	PID	before(P)	after(P)	prevLSN
237	T3	P15	...	...	...
238	T2	P18	...	...	...
239	T1	P17	...	...	...
240	T1	COMMIT			...
241	T2	COMMIT			...
242		CKP			
243	T3	P19	...	...	...
244	T3	COMMIT			...

# Media failure

---

- Nel caso di media failure si ha un ripristino che usa una copia archiviata del DB (**DataBase Dump**)
- Facendo uso del Log, si rifanno quindi tutte le transazioni che hanno eseguito **COMMIT** e si dismano tutte quelle per cui il record di **COMMIT** non si trova nel Log
- È evidente che se il Log dovesse corrompersi tutte le operazioni di ripristino non potrebbero funzionare correttamente
- Per tale motivo è comune mantenere il Log almeno in **duplice copia**

# Transazioni in MySQL

---

- In MySQL (InnoDB) si adotta il modello **Steal/No-force**, per cui è possibile che sia necessario fare sia UNDO che REDO di transazioni
- Il modello di **checkpoint** è *fuzzy*, ovvero la scrittura di tutte le pagine modificate avviene in batch di ridotte dimensioni, per interferire meno con la normale attività del sistema
- In ogni caso il checkpoint viene eseguito quando i file di log si riempiono
- Es: `innodb_log_file_size=200M` crea 2 log file (`ib_logfile0` e `ib_logfile1`) di 200M l'uno



# Transazioni in PostgreSQL

---

- PostgreSQL permette un controllo fine sul protocollo WAL, considerando anche l'interazione con il sistema operativo sottostante
- Esempi:
  - `fsync(boolean)`: abilita/disabilita la `fsync()` system call per forzare la scrittura su disco del log di transazioni committed
  - `commit_delay(integer)`: ritardo tra la scrittura del commit record in un log buffer e la scrittura del buffer stesso su disco (per risparmiare `fsync()`)
- Dettagli: capp. 17.5 e 27

# Elaborazione di interrogazioni

---

# Passi del processo di elaborazione

---

- L'elaborazione di una query SQL comporta l'esecuzione di una serie di passi, i cui principali sono:
  1. La fase di **parsing** verifica che la query sia scritta correttamente
  2. La fase di **check semantico** verifica che gli oggetti referenziati esistano, ecc.
  3. Il **rewriter** modifica la query riscrivendola, ad esempio sostituendo alle viste la loro definizione
  4. Il **planner** (o **optimizer**) considera modalità alternative di esecuzione e sceglie quella reputata migliore, generando un **query plan**
  5. L'**executor** esegue il query plan
- Ovviamente, ogni DBMS adotta soluzioni specifiche

# Check semantico

---

- La fase di **check semantico** verifica, tra le altre cose, che:
  - Gli oggetti referenziati esistano
  - Gli operatori siano applicati a dati di tipo opportuno
  - L'utente abbia i privilegi necessari per eseguire l'operazione
- A tale scopo **si fa riferimento ai cataloghi**, in cui sono memorizzate tutte le informazioni necessarie

# Check semantici e uso dei cataloghi (1)

---

- PostgreSQL ha per ogni DB, un `information_schema` (cap. 32) che contiene viste definite in SQL standard, e cataloghi propri (table e view) (cap. 43), tra cui:

Catalog name	Attribute	Descrizione
pg_namespace	oid	OID dello schema
pg_namespace	name	Nome dello schema
pg_class	oid	OID della classe (table, view, indice, ...)
pg_class	relnamespace	OID dello schema
pg_class	relname	Nome della table, vista o indice
pg_class	relowner	OID del proprietario
pg_attribute	attrelid	OID della classe
pg_attribute	attname	Nome dell'attributo

# Check semantici e uso dei cataloghi (2)

---

- Supponiamo di avere la seguente query:

```
SELECT    EmpNo
FROM      MySchema.Employee
```

- In fase di check semantico vengono eseguite query del tipo:

```
SELECT    *
FROM      pg_class C, pg_namespace NS
WHERE     C.relnamespace = NS.oid
          AND NS.name = 'MySchema'
          AND C.relname = 'Employee' ;
```

```
SELECT    *
FROM      pg_class C, pg_namespace NS, pg_attribute A
WHERE     C.relnamespace = NS.oid
          AND NS.name = 'MySchema'
          AND C.relname = 'Employee'
          AND C.oid = A.attrelid
          AND A.attname = 'EmpNo' ;
```

# Riscrittura di interrogazioni

---

- Prima di procedere alla fase vera e propria di ottimizzazione della query, il DBMS esegue un passo di “**rewriting**” della stessa
- Lo scopo della fase di riscrittura è **semplificare la query e pervenire a una forma più semplice da analizzare e quindi ottimizzare**
- Tra le operazioni tipiche che hanno luogo in questa fase vi sono:
  - **Risoluzione delle viste**: si esegue il merge della query in input con le query che definiscono le viste referenziate
  - **Unnesting**: se la query include delle subquery si prova a trasformarla in una forma senza innestamenti
  - **Uso dei vincoli**: vengono sfruttati i vincoli definiti sugli schemi al fine di semplificare la query
- Il modo con cui vengono eseguite queste operazioni varia da sistema a sistema, e quindi non è possibile fornire soluzioni di validità generale (il modo cambia anche per uno stesso DBMS se si scelgono “**parametri di ottimizzazione**” differenti)

# Risoluzione di viste

---

- Si consideri la vista:

```
CREATE VIEW EmpSalaries(EmpNo,Last,First,Salary)
AS  SELECT  EmpNo,LastName,FirstName,Salary
    FROM    Employee
    WHERE   Salary > 20000
```

e la query: 

```
SELECT  Last, First
FROM    EmpSalaries
WHERE   Last LIKE 'B%'
```

- La risoluzione della vista porta a riscrivere la query come:

```
SELECT  LastName,FirstName
FROM    Employee
WHERE   Salary > 20000
    AND  LastName LIKE 'B%'
```



# Unnesting

---

- Il passaggio a una forma senza subquery alle volte è di immediata comprensione; ad esempio la seguente query:

```
SELECT  EmpNo, PhoneNo
FROM    Employee
WHERE   WorkDept IN ( SELECT  DeptNo
                       FROM    Department
                       WHERE   DeptName = 'Operations' )
```

viene riscritta come:

```
SELECT  EmpNo, PhoneNo
FROM    Employee E, Department D
WHERE   E.WorkDept = D.DeptNo
        AND  D.DeptName = 'Operations'
```

# Uso dei vincoli

---

- In generale, la presenza di vincoli può portare a semplificare le query.
- Ad esempio, se **EmpNo** è una chiave:

```
SELECT DISTINCT EmpNo  
FROM Employee
```

si riscrive più semplicemente come:

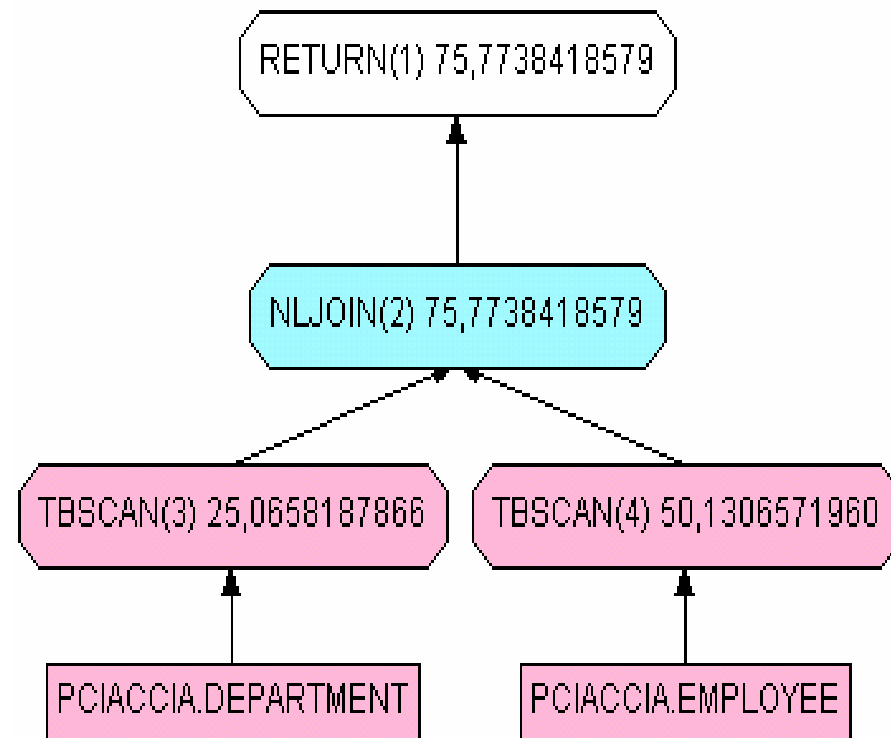
```
SELECT EmpNo  
FROM Employee
```

che ha il vantaggio di non comportare una (inutile!) operazione di rimozione delle tuple duplicate dal risultato

# Piani di accesso

- Al termine della fase di riscrittura viene eseguita l'ottimizzazione vera e propria, anche detta “**ottimizzazione basata sui costi**”
- In questa fase, facendo anche uso delle informazioni **statistiche sui dati**, viene scelto il modo “più economico” per eseguire la query, ovvero il modo che complessivamente presenta un costo minimo tra tutte le alternative possibili
- Ogni “modo” di eseguire una query è detto **piano di accesso**, e si compone di una serie di operatori connessi ad albero
- Le **foglie** del piano di accesso sono le **relazioni di base** presenti nella query riscritta
- **Gli altri nodi sono operatori che agiscono su 1 o 2 insiemi di tuple in input e producono 1 insieme di tuple in output**

```
SELECT *  
FROM Department, Employee  
WHERE WorkDept = DeptNo
```



# Operatori relazionali

---

- Parlando di operatori è opportuno distinguere tra:
  - **Operatori logici** (es. Join, Sort)
    - Svolgono una determinata funzione e producono un insieme di tuple con certe proprietà (ad es. ordinate su un campo)
  - **Operatori fisici** (es. join nested-loops, o NLJOIN)
    - Sono implementazioni specifiche di un operatore logico; in funzione dei valori delle statistiche e dei parametri di sistema (es. dimensione del buffer pool) è possibile associare ad ogni operatore fisico un **costo (stimato) di esecuzione**

# L'operatore Sort

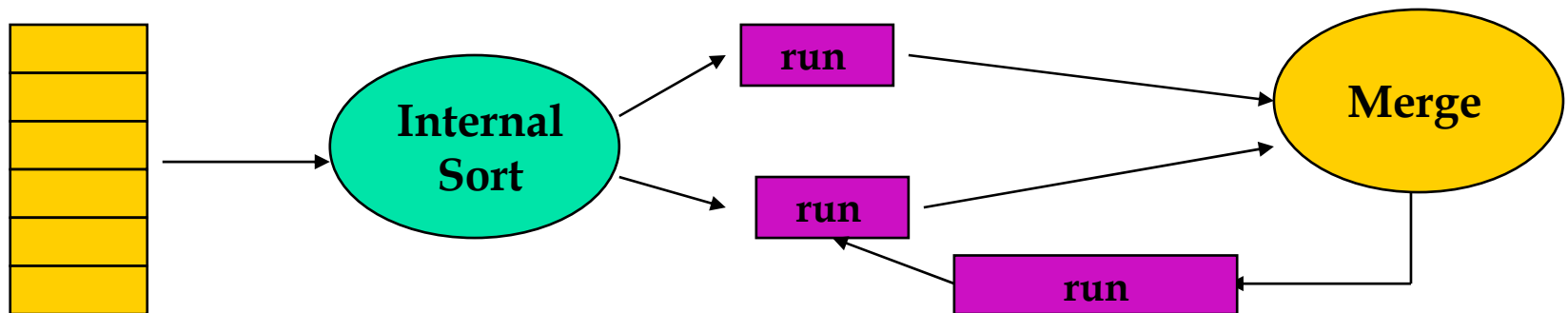
---

- L'operatore Sort, nella sua versione di base, riceve in input un insieme di tuple e produce in output lo stesso insieme di tuple ordinato su un attributo A (o su una combinazione di attributi  $A_1, A_2, \dots, A_n$ )
- Esistono ovviamente diverse varianti: ad esempio, se alcuni attributi in input non servono nell'output, si possono eliminare durante l'esecuzione
- Dal punto di vista implementativo, l'algoritmo più comunemente utilizzato dai DBMS è quello detto di **Merge Sort a Z vie** (**Z-way Sort-Merge** o anche **External Merge Sort**), in grado di ordinare input la cui dimensione eccede quella della memoria a disposizione

# Z-way Sort-Merge

---

- Dato un input di  $NP$  pagine e un numero  $NB < NP$  di buffer in memoria centrale, l'algoritmo opera in **2 fasi**:
  - **Sort interno**: ordina l'input a blocchi di  $NB$  pagine, producendo quindi delle "run" ordinate che vengono scritte su disco in un file temporaneo
  - **Merge**: opera uno o più passi di fusione, "fondendo" le run iniziali in run di dimensione maggiore, fino a produrre un'unica run



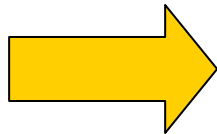
# Z-way Merge Sort: esempio

- Supponiamo di avere a disposizione solo  $NB = 2$  buffer per leggere l'input

3 record per pagina

g	24
a	19
d	21
c	33
b	14
e	16
r	16
d	31
m	3
p	2
d	7
a	14

Input



Sort interno

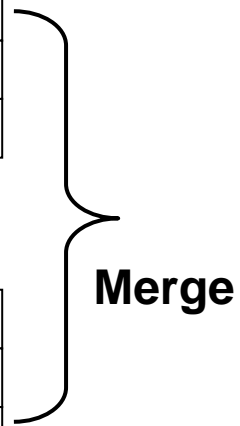
a	19
b	14
c	33
d	21
e	16
g	24

run1

a	14
d	7
d	31
m	3
p	2
r	16

run2

2-page runs



a	14
a	19
b	14
c	33
d	7
d	21
d	31
e	16
g	24
m	3
p	2
r	16

Output

In fase di Merge, in ogni istante in memoria si ha una pagina di run1 e una di run2

Questo è sufficiente per fondere ordinatamente le 2 run

# Utilità del Sort

---

- Oltre che per ordinare le tuple il Sort può essere utilizzato per:
  - Query in cui compare l'opzione **DISTINCT**, ovvero per eliminare i duplicati
  - Query che contengono la clausola **GROUP BY**
- In entrambi i casi alcuni DBMS fanno uso di operatori che usano **tecniche hash** per “raggruppare” le tuple con valori uguali su tutti gli attributi (nel caso di **DISTINCT**) o solo su alcuni (nel caso di **GROUP BY**)
- Il Sort può anche essere usato come passo preliminare per eseguire un **Merge Scan Join**



# L'operatore Join

---

- L'operatore Join, nella sua versione di base, ha come
  - INPUT:** due insiemi di tuple
  - OUTPUT:** un insieme in cui ogni tupla è ottenuta combinando, sulla base di uno o più predicati di join, una tupla del primo insieme con una tupla del secondo insieme
- Anche per il Join vi sono diverse varianti:
  - Nel caso di **Outer Join** non è richiesta la presenza di “matching tuples”
  - Gli attributi non richiesti in output possono essere eliminati quando si produce il risultato del Join

```
SELECT      E.Empno, D.DeptNo
FROM        Employee E, Department D
WHERE       E.WorkDept = D.DeptNo
           AND      E.EmpNo <> D.MgrNo
```

# Operatori fisici per il Join

---

- Esistono moltissime implementazioni del Join, che mirano a sfruttare al meglio le risorse del sistema e le (eventuali) proprietà degli insiemi di tuple in ingresso per evitare di eseguire tutti i possibili confronti tra le tuple
- Le implementazioni più diffuse si riconducono ai seguenti operatori fisici:
  - **Nested Loops Join** (NLJOIN)
  - **Merge Scan Join** (MSJOIN)
  - **Hash Join** (HSJOIN)
- Si noti che, benché logicamente il Join sia commutativo, dal punto di vista fisico vi è una chiara distinzione, che influenza anche le prestazioni, tra operando sinistro (o “**esterno**”, “outer”) e operando destro (o “**interno**”, “inner”)

# Nested Loops Join

- Date 2 relazioni in input **R** e **S** tra cui sono definiti i predicati di join **PJ**, e supponendo che **R** sia la **relazione esterna**, l'algoritmo opera come segue

**Per ogni** tupla  $t_R$  in **R**:

{ **Per ogni** tupla  $t_S$  in **S**:

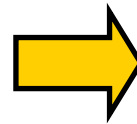
{ **se** la coppia  $(t_R, t_S)$  soddisfa **PJ**

**allora** aggiungi  $(t_R, t_S)$  al risultato } }

R	A	B
	22	a
	87	s
	45	h
	32	b

**PJ: R.A = S.A**

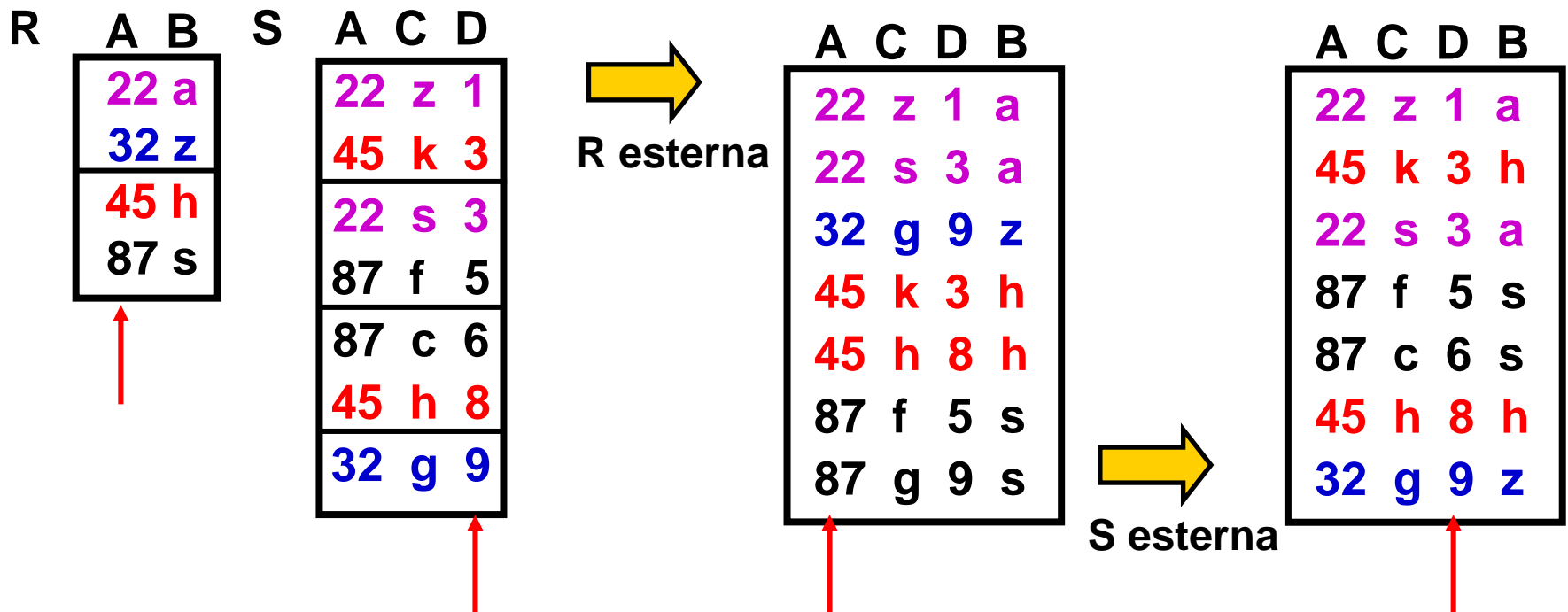
S	A	C	D
	22	z	8
	45	k	4
	22	s	7
	87	s	9
	32	c	3
	45	h	5
	32	g	6



A	C	D	B
22	z	8	a
22	s	7	a
87	s	9	s
45	k	4	h
45	h	5	h
32	c	3	b
32	g	6	b

# Nested loops: cammini di accesso

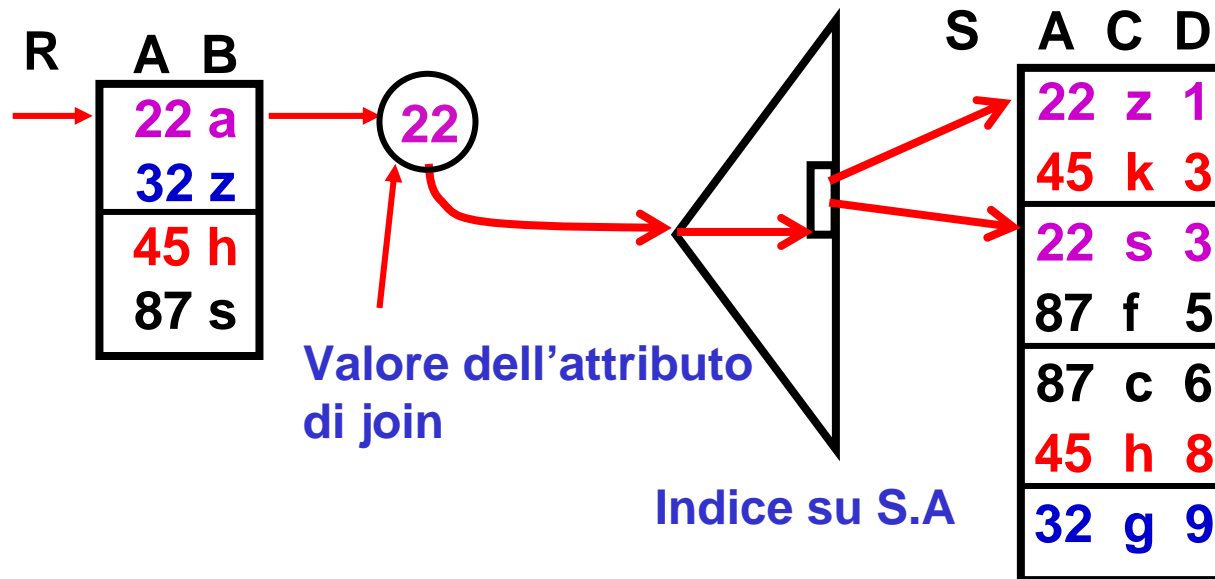
- L'ordine con cui vengono generate le tuple del risultato coincide con l'ordine eventualmente presente nella relazione esterna



Pertanto se l'ordine che si genera è "interessante", ad esempio perché la query contiene **ORDER BY S.D**, la scelta della relazione esterna può risultarne influenzata (si risparmia lavoro in seguito!)

# Nested loops: presenza di indici

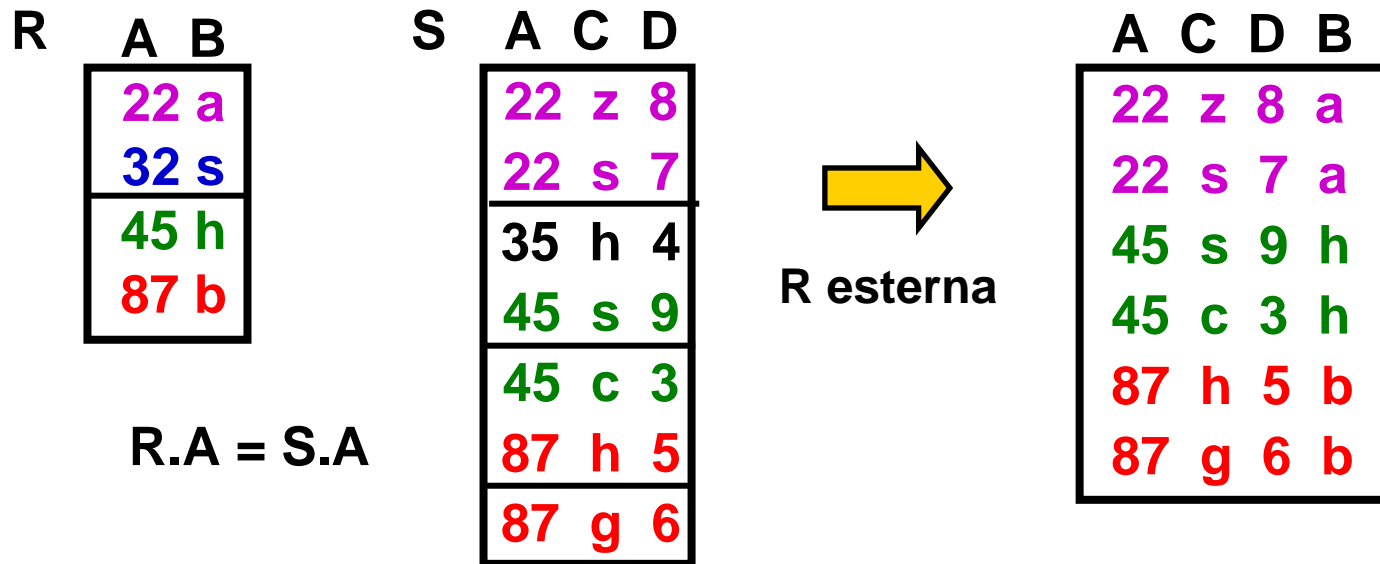
- Data una tupla della relazione esterna R, la scansione completa della relazione interna S può essere sostituita da una scansione basata su un indice costruito sugli attributi di join di S, secondo il seguente schema:



- L'accesso alla relazione interna mediante indice porta in generale a ridurre di molto i costi di esecuzione del Nested Loops Join

# Merge Scan Join

- E' applicabile quando **gli insiemi di tuple in input sono ordinati sugli attributi di join**
- Per R (S) ciò è possibile se R (S) è fisicamente ordinata sugli attributi di join oppure se esiste un indice sugli attributi di join di R (S)



- Se non è possibile accedere ordinatamente a uno o entrambi gli insiemi in input, **è sempre possibile eseguirne il Sort**
- Il costo di esecuzione aumenta, ma può risultare minore di quello di un Nested Loops

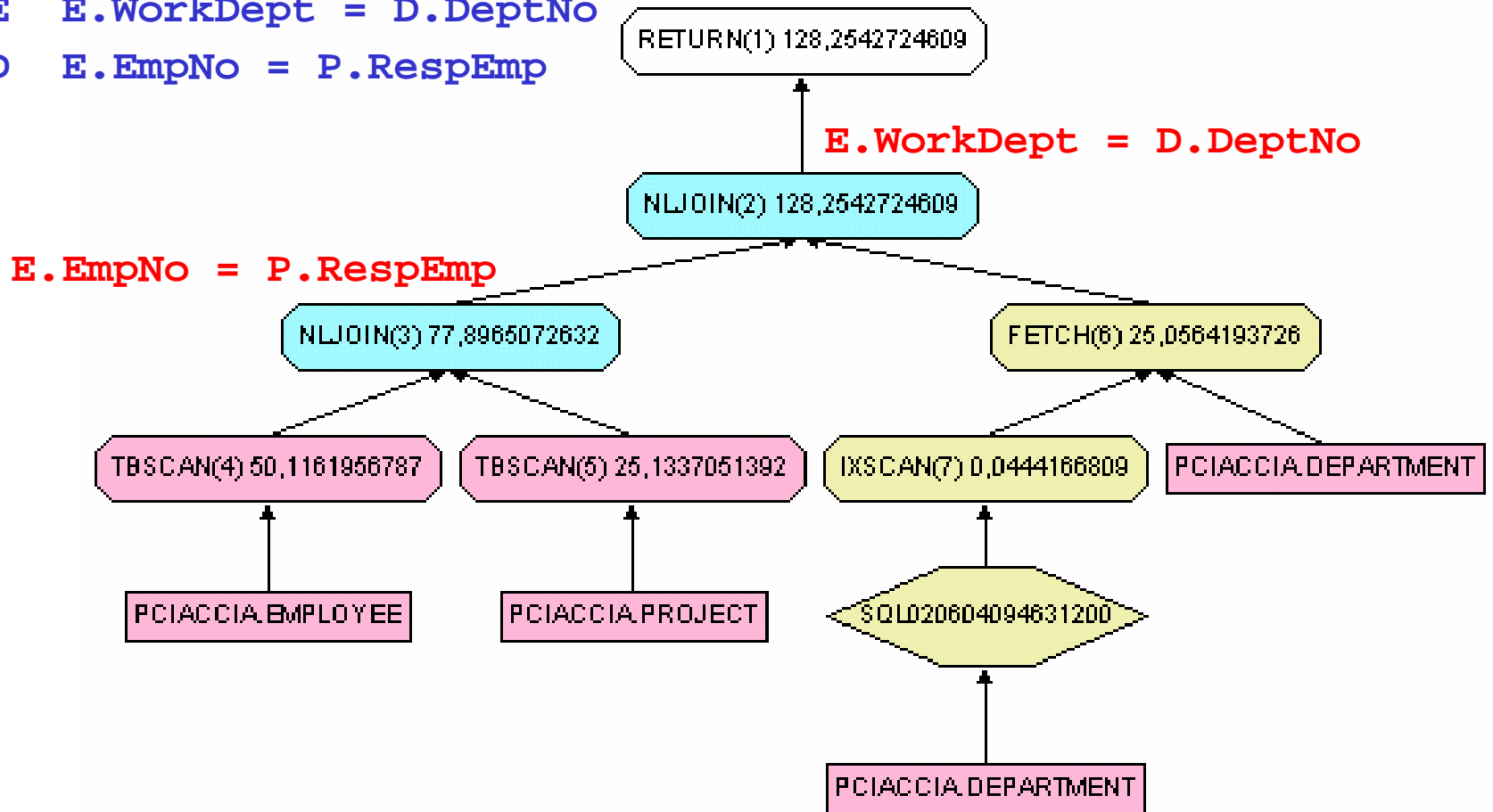
# Valutazione per materializzazione

---

- Un semplice modo di eseguire un piano di accesso composto da diversi operatori consiste nel procedere **bottom-up**, secondo il seguente schema:
  - Si calcolano innanzitutto i risultati degli operatori al livello più basso dell'albero e si memorizzano tali risultati in **relazioni temporanee**
  - Si procede quindi in modo analogo per gli operatori del livello sovrastante, fino ad arrivare alla radice
- Tale modo di procedere, detto “**valutazione per materializzazione**”, è **altamente inefficiente**, in quanto comporta la creazione, scrittura e lettura di molte relazioni temporanee, relazioni che, se la dimensione dei risultati intermedi eccede lo spazio disponibile in memoria centrale, devono essere gestite su disco

# Materializzazione: esempio (1)

```
SELECT P.ProjNo, E.EmpNo, D.*  
FROM Department D,Employee E,Project P  
WHERE E.WorkDept = D.DeptNo  
AND E.EmpNo = P.RespEmp
```





# Materializzazione: esempio (2)

- La valutazione per materializzazione produrrebbe come risultato del primo Join (**E.EmpNo = P.RespEmp**):

PROJNO	EMPNO	WORKDEPT
AD3100	000010	A00
MA2100	000010	A00
PL2100	000020	B01
IF1000	000030	C01
IF2000	000030	C01
OP1000	000050	E01
OP2000	000050	E01
MA2110	000060	D11
AD3110	000070	D21
OP1010	000090	E11
OP2010	000100	E21
MA2112	000150	D11
MA2113	000160	D11
MA2111	000220	D11
AD3111	000230	D21
AD3112	000250	D21
AD3113	000270	D21
OP2011	000320	E21
OP2012	000330	E21
OP2013	000340	E21

- A partire da tale risultato intermedio si può poi calcolare il Join **E.WorkDept = D.DeptNo**

PROJNO	EMPNO	DEPTNO	DEPTNAME	MGRNO	ADMRDEPT	LOCATION
AD3100	000010	A00	SPIFFY CO...	000010	A00	
MA2100	000010	A00	SPIFFY CO...	000010	A00	
PL2100	000020	B01	PLANNING	000020	A00	
IF1000	000030	C01	INFORMAT...	000030	A00	
IF2000	000030	C01	INFORMAT...	000030	A00	
OP1000	000050	E01	SUPPORT ...	000050	A00	
OP2000	000050	E01	SUPPORT ...	000050	A00	
MA2110	000060	D11	MANUFAC...	000060	D01	
AD3110	000070	D21	ADMINIST...	000070	D01	
OP1010	000090	E11	OPERATIO...	000090	E01	
OP2010	000100	E21	SOFTWARE...	000100	E01	
MA2112	000150	D11	MANUFAC...	000060	D01	
MA2113	000160	D11	MANUFAC...	000060	D01	
MA2111	000220	D11	MANUFAC...	000060	D01	
AD3111	000230	D21	ADMINIST...	000070	D01	
AD3112	000250	D21	ADMINIST...	000070	D01	
AD3113	000270	D21	ADMINIST...	000070	D01	
OP2011	000320	E21	SOFTWARE...	000100	E01	
OP2012	000330	E21	SOFTWARE...	000100	E01	
OP2013	000340	E21	SOFTWARE...	000100	E01	

# Valutazione in pipeline

- Un modo alternativo di eseguire un piano di accesso è quello di **eseguire più operatori in pipeline**, ovvero **non aspettare che termini l'esecuzione di un operatore per iniziare l'esecuzione di un altro**

- Nell'esempio precedente, la valutazione in pipeline opererebbe così:

- Si inizia a eseguire il primo Join (**E.EmpNo = P.RespEmp**). Appena

viene prodotta la prima tupla dell'output: questa viene passata in input al

PROJNO	EMPNO	WORKDEPT
AD3100	000010	A00

secondo Join (**E.WorkDept = D.DeptNo**), che può quindi iniziare la ricerca di matching tuples e quindi produrre la prima tupla del risultato finale della query:

PROJNO	EMPNO	DEPTNO	DEPTNAME	MGRNO	ADMRDEPT	LOCATION
AD3100	000010	A00	SPIFFY CO...	000010	A00	

- La valutazione prosegue cercando eventuali altri match per la tupla prodotta dal primo Join; quando è terminata la scansione della relazione interna (**Department**), il secondo Join richiede al primo Join di produrre un'altra tupla, ecc.

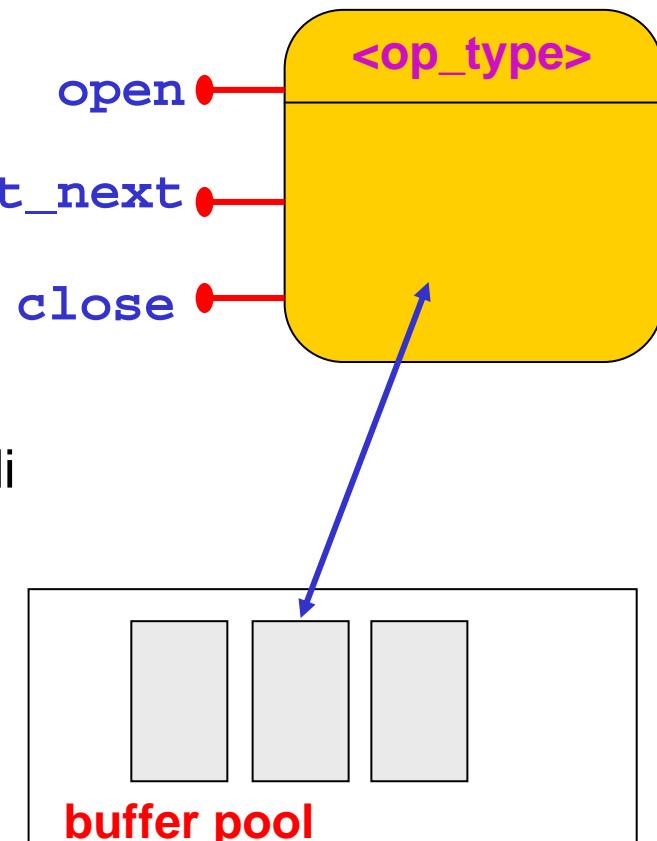
# Interfaccia a iteratore

- Per supportare una valutazione in pipeline e per semplificare la realizzazione dei diversi operatori, i DBMS definiscono gli operatori mediante un'interfaccia a "iteratore", i cui metodi principali sono:

**open**: inizializza lo stato dell'operatore, alloca buffer per gli input e l'output, richiama ricorsivamente **open** sugli operatori figli; viene anche usato per passare argomenti (ad es. la condizione che un operatore **Filter** deve applicare)

**get\_next**: usato per richiedere un'altra tupla del risultato dell'operatore; l'implementazione di questo metodo include **get\_next** sugli operatori figli e codice specifico dell'operatore

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



# Pipeline con iteratori

---

- L'interfaccia a iteratore supporta naturalmente una esecuzione in pipeline degli operatori, in quanto **la decisione se lavorare in pipeline o materializzare è incapsulata nel codice specifico dell'operatore**
  - Se la logica dell'operatore permette di elaborare completamente una tupla in input appena questa viene ricevuta, allora l'input non viene materializzato e si può procedere in pipeline
    - E' questo il caso del Nested Loops Join
  - Se, viceversa, la logica dell'algoritmo richiede di esaminare la stessa tupla in input più volte, allora si rende necessario materializzare
    - E' questo il caso del Sort, che non può produrre in output una tupla senza prima aver esaminato tutte le altre
- E' importante osservare che **l'interfaccia a iteratore viene usata anche per incapsulare metodi di accesso quali i B<sup>+</sup>-tree**, che esternamente vengono quindi visti semplicemente come operatori che producono un insieme (**stream**) di tuple

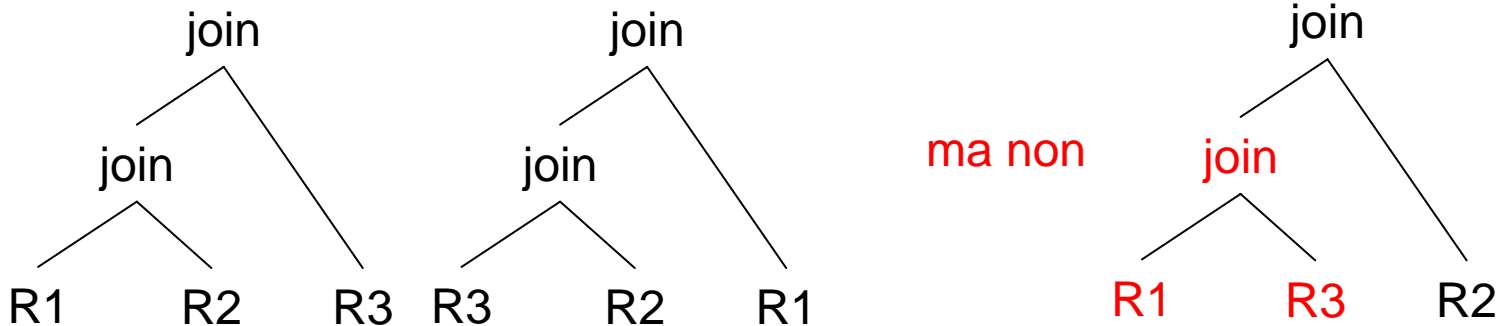
# Determinazione del piano ottimale

---

- L'ottimizzatore, al fine di poter determinare il piano di accesso a costo minimo, dispone di una **strategia di enumerazione (generazione) dei piani di accesso**, i cui compiti principali sono:
  - **Enumerare tutti i piani che, potenzialmente, possono risultare ottimali**
  - **Non generare piani di accesso che sicuramente non possono risultare ottimali**
- Il numero di piani di accesso che vengono generati per ottimizzare una query può risultare molto elevato
  - Ad esempio, se una query esegue il **join di N relazioni**, esistono **almeno  $N! = 1 * 2 * \dots * N$  piani di accesso potenzialmente ottimali** che l'ottimizzatore deve considerare
- Diversi DBMS permettono di controllare esplicitamente la strategia di enumerazione, in modo da permettere un "tuning" più fine delle prestazioni

# L'ottimizzatore di PostgreSQL

- PostgreSQL segue fedelmente il modello descritto, in particolare:
  - Implementa i 3 operatori fisici per il join
  - Genera (quasi) tutti i possibili piani di accesso
    - Esclude quelli in cui si combinano relazioni senza predicati di join
    - Es: per  $R1.A = R2.A \text{ AND } R2.B = R3.B$  considera ad es.



- Nel caso di query con molti join (default: > 11), permette di passare a un'ottimizzazione approssimata basata su [algoritmi genetici](#) (**GEQO** – Genetic Query Optimization, cap. 48)
- Come influenzare l'ottimizzatore: cap. 17.6
- **ANALYZE** aggiorna le statistiche

# L'ottimizzatore di MySQL

---

- Le regole di ottimizzazione di MySQL sono tuttora in evoluzione, e implementano una combinazione di ricerca esaustiva e di tecniche euristiche ad hoc (cap. 7.2)
- Esempi:
  - se in una query con `ORDER BY` tutte le colonne di ordinamento sono di R1, allora R1 sarà sempre la prima relazione ad essere considerata per eseguire i join
  - Non viene eseguito l'unnesting delle query, ma la query viene comunque riscritta
- `EXPLAIN SELECT` fornisce informazioni dettagliate su come una query viene eseguita
- `EXPLAIN [VERBOSE]` è l'analogo in PostgreSQL

# DB attivi: trigger e regole

---



# Trigger

---

- I trigger sono particolari regole di tipo **ECA** (**E**vent, **C**ondition, **A**ction):  
Un trigger si attiva a fronte di un dato evento e, se sussiste una data condizione, allora esegue una data azione

```
CREATE TRIGGER EmpSalaryTooHigh
AFTER UPDATE OF Salary ON Employee           -- evento
FOR EACH ROW
WHEN (NEW.Salary >                            -- condizione
      SELECT Salary FROM Employee
      WHERE EmpCode = NEW.EmpManager)
UPDATE Employee                               -- azione
SET Salary = OLD.Salary
WHERE EmpCode = NEW.EmpCode
```

- **NEW** permette di riferirsi ai nuovi valori, **OLD** a quelli vecchi

# Trigger: modalità di base

---

- Un trigger può attivarsi prima (**BEFORE**) o dopo (**AFTER**) l'evento corrispondente: i primi “condizionano” l'operazione di modifica, i secondi “reagiscono” alla modifica stessa
- Un trigger può essere definito a livello di singola tupla (**FOR EACH ROW**) o di istruzione SQL (**FOR EACH STATEMENT**). In questo secondo caso il trigger opera a livello aggregato
  - Es.: controllo dello stipendio medio

```
CREATE TRIGGER CheckSalaries
AFTER UPDATE OF Salary ON Employee
FOR EACH STATEMENT      -- default
WHEN ((SELECT AVG(Salary) FROM Employee) > 1000)
UPDATE Employee
SET Salary = 0.9*Salary
```

# Trigger: cicli

---

- In generale ragionare con i trigger è complesso, in quanto si possono avere attivazioni in cascata e anche cicli infiniti

```
CREATE TRIGGER T1
AFTER INSERT INTO R
FOR EACH ROW
DELETE FROM R
WHERE id = NEW.id
```

```
CREATE TRIGGER T2
AFTER DELETE FROM R
FOR EACH ROW
INSERT INTO R
VALUES (OLD.id, ...)
```

anche con un solo trigger(!):

```
CREATE TRIGGER CheckSalariesWrong
AFTER UPDATE OF Salary ON Employee
WHEN ((SELECT AVG(Salary) FROM Employee) > 1000)
UPDATE Employee
SET Salary = 1.1*Salary
```

# Trigger in MySQL e PostgreSQL

---

- Dalla versione 5.0.2 MySQL include supporto per i trigger (cap. 18), con qualche limitazione (es.: impossibilità di definire una transazione nella parte di azione, manca la modalità **FOR EACH STATEMENT**)
- Viceversa, PostgreSQL, benché permetta la definizione di trigger (cap. 34), ne ha una gestione limitata, in particolare non ha riferimenti **NEW** e **OLD**, e la parte di azione deve essere scritta definendo una funzione (**CREATE FUNCTION**) scritta in un linguaggio arbitrario (SQL, C, ecc.)
- Per contro, PostgreSQL mette a disposizione un potente sistema di **regole** che è integrato nell'elaborazione delle istruzioni SQL

# Regole in PostgreSQL

---

- Il sistema di **regole** di PostgreSQL opera a valle della fase di parsing e a monte di quella di ottimizzazione
- Di fatto, PostgreSQL usa le regole per svariati scopi, ad esempio per risolvere le viste:

```
CREATE VIEW myview AS SELECT * FROM mytab;
```

è realizzata come

```
CREATE TABLE myview (same column list as mytab);  
CREATE RULE "_RETURN" AS ON SELECT  
TO myview  
DO INSTEAD SELECT * FROM mytab;
```

- La sintassi generale è:

```
CREATE RULE name AS ON event  
TO table [ WHERE condition ]  
DO [ALSO|INSTEAD] {NOTHING|command| (command;command... )}
```

# Estendibilità

---

# Cos'è l'estendibilità

---

- Parlare di estendibilità per un DBMS significa poter aggiungere allo stesso:
  - Nuovi tipi di dato (UDT: User Defined Type)
  - Nuove funzioni (UDF: User Defined Function)
  - Nuovi aggregati
  - Nuovi operatori
  - Nuovi tipi di indice
  - ...
- garantendo l'integrazione a tutti i livelli
- PostgreSQL, sin dalle sue origini, ha considerato l'estendibilità un requisito base, per poter far fronte alle nuove esigenze di gestione dati
  - Il tutto gestito mediante i cataloghi (cap. 33)
- Nel tempo, questa tendenza si è diffusa anche nei sistemi commerciali, che hanno messo a punto tecnologie proprietarie (extender, datablade, ecc.) per gestire, ad esempio, dati geografici e multimediali

# Nuove Funzioni in MySQL

---

- Esistono 2 modi, senza dover modificare il server:
  - UDF: la f. viene compilata come object file e collegata dinamicamente:

```
CREATE FUNCTION function_name
RETURNS {STRING | INTEGER | REAL | DECIMAL}
SONAME shared_library_name
```
  - Stored routine: la funzione è scritta in SQL e viene definita completamente mediante CREATE FUNCTION

```
CREATE FUNCTION sp_name ([func_parameter[,...]])
RETURNS type
characteristic [...] routine_body
```

    - *characteristic*: LANGUAGE SQL | [NOT] DETERMINISTIC  
| { CONTAINS SQL | NO SQL | READS SQL DATA |  
MODIFIES SQL DATA }
    - *routine\_body*: Valid SQL procedure statement
- Dettagli: capp. 24.2 e 17



# Nuove Funzioni in PostgreSQL

---

- In maniera simile, esistono diversi modi di definire nuove funzioni in PostgreSQL:
  - SQL function: consiste di uno o più statement SQL, ad es.:

```
CREATE FUNCTION clean_emp() RETURNS void AS  
'DELETE FROM emp WHERE salary < 0;' LANGUAGE SQL;  
SELECT clean_emp();
```
  - Funzioni in una shared library:

```
CREATE FUNCTION sp_name ([func_parameter[,...]])  
RETURNS type  
characteristic ...] routine_body
```

    - characteristic: LANGUAGE SQL | [NOT] DETERMINISTIC  
| { CONTAINS SQL | NO SQL | READS SQL DATA |  
MODIFIES SQL DATA }
    - routine\_body: Valid SQL procedure statement
- Dettagli: capp. 24.2 e 17

# PostgreSQL: UDT

---

- In PostgreSQL esistono 2 modalità base per creare un nuovo tipo di dato:
  - Tipo di dato composto: corrisponde a definire il formato di un record, ad es:  

```
CREATE TYPE INTTEXT AS (A INT, B TEXT)
```

e viene tipicamente usato per rappresentare l'input/output di una funzione
  - Tipo di dato base: in questo caso è necessario specificare delle funzioni di conversione (lettura, scrittura) e altri parametri necessari a interpretare correttamente i nuovi valori