

Lezioni del Prof. Pellegrini

INDICE

Il DBMS	1
Le applicazioni... hanno uno stato	1
<i>Dati strutturati?</i>	1
<i>DBMS: accesso ai dati e manipolazione della struttura</i>	2
Le applicazioni... sono distribuite sulla rete	2
<i>Livelli di architettura</i>	3
Le applicazioni... sono parallele	4
<i>Problemi legati all'uso dei lock</i>	4
<i>Composizione dei lock</i>	4
Interrogazioni al DBMS	6
<i>Object Relational Mapping (ORM)</i>	6
JDBC	6
<i>java.sql.Connection</i>	7
<i>java.sql.Statement</i>	7
<i>java.sql.ResultSet</i>	7
<i>Data Access Object (DAO) Pattern</i>	7
Organizzazione fisica	9
<i>DBMS e file system</i>	9
Pagine e record	10
<i>Pagine con record di lunghezza fissa e variabile</i>	10
<i>Formato di un record a taglia fissa e variabile</i>	11
Organizzazione dei file	12
<i>Heap file</i>	12
<i>File con pagine ordinate</i>	13
<i>Hashed file</i>	13
Modello di costo (tempo di esecuzione)	13
<i>Istanziamento del modello per heap file</i>	14
<i>Istanziamento del modello per file ordinato</i>	14
<i>Istanziamento del modello per file hash (approssimata)</i>	15
File indicizzati	16
<i>Struttura di una entry dati</i>	16

<i>Raggruppato/non raggruppato</i>	17
<i>Indici primari e secondari</i>	17
<i>Indici sparsi e densi</i>	17
<i>Chiavi semplici o composite</i>	18
Indici e organizzazioni degli indici	18
<i>Indice ad albero</i>	18
<i>Indexed Sequential Access Method (ISAM)</i>	19
<i>Indice B⁺-tree</i>	21
<i>k-d Trees</i>	23
<i>R-Tree</i>	23
SQL avanzato	24
<i>Vincoli di integrità generici: check e asserzioni</i>	24
<i>Basi di dati attive: i trigger</i>	24
<i>SQL States</i>	26
<i>Creazione di indici</i>	27
<i>Eventi temporizzati</i>	27
<i>Funzioni</i>	27
<i>Creazione Stored Procedure</i>	28
<i>Livelli di Isolamento</i>	29
<i>Modalità di accesso transazionali</i>	30
<i>Come marcare le transazioni</i>	30
<i>Cursori</i>	30
Prepared statement	32
<i>Gestione di prepared statement</i>	32
<i>Invocazione di Stored Procedure</i>	33

Il DBMS

Il DBMS è quel componente software che viene spesso (impropriamente) chiamato *database*:

- DBMS: applicativo per la gestione dei dati,
- database: “banca dati”, ossia i dati strutturati manipolati dal DBMS.

L'importanza dei DBMS nasce dal fatto che essi permettono di raggiungere un *livello di astrazione* maggiore rispetto a molte caratteristiche ricorrenti delle applicazioni. In tal senso, essi permettono:

- uno sviluppo più veloce (non necessità di reinventare la ruota),
- un dispiegamento più performante (ottimizzazioni di performance e beneficio di tutti gli utilizzatori del DBMS).

Le applicazioni... hanno uno stato

La **persistenza** è un concetto legato alla caratteristica dei dati di un programma di sopravvivere all'esecuzione del programma stesso. È un concetto fondamentale: allo spegnimento della macchina il contenuto della RAM è perduto ed è possibile ottenere persistenza soltanto affidandosi a dispositivi non volatili: dischi, nastri, memoria non volatile.

Dati strutturati?

Nella maggior parte dei casi, i programmi informatici trattano *dati strutturati*: come possiamo organizzarli? Un possibile esempio è il seguente:

```
struct book {  
    char title[10];  
    char author[10];  
    int publication_year;  
    float price;  
};
```

Di per sé, una struct altro non è che un'organizzazione strutturata dei dati, in cui data la definizione della struttura sappiamo esattamente quanti dati dobbiamo leggere, dove e così via.

Alternativamente, è possibile ricorrere alla **programmazione a oggetti**.

Supponiamo di voler costruire il nostro primo database in C. Quello che andremo a scrivere in memoria non sarà più un flusso di dati, ma un record; quindi, possiamo dire che il nostro database non è che un file in cui abbiamo scritto dei dati strutturati, la cui organizzazione ci rende più facile l'accesso ed il lavoro sugli stessi.

Nel nostro file possiamo scrivere quanti record vogliamo, ma occorre rispondere ad alcune domande:

1. in che ordine memorizziamo i record?
2. come effettuiamo operazioni di aggiunta di record?
3. come effettuiamo operazioni di ricerca di record?

Sono tutte domande le cui risposte sono *algoritmiche* e c'è quindi la necessità di implementare da capo queste funzionalità in ciascuna applicazione. Inoltre, vi è un altro problema: non è detto che l'organizzazione dei dati rimanga immutata nel tempo! Di conseguenza, potremmo dover andare a modificare il codice ad hoc che avevamo scritto in precedenza, perdendo ulteriore (e prezioso) tempo.

Infatti, per poter incorporare un'ipotetica modifica alla struttura delle informazioni è necessario:

- rileggere tutto il database,
- riscrivere tutto il database,
- aggiornare il software,
- ricompilare/ridistribuire il database.

Inoltre, i tempi e i costi possono non essere minimali.

DBMS: accesso ai dati e manipolazione della struttura

Il DBMS offre un'astrazione di livello elevato per la manipolazione delle informazioni:

```
INSERT INTO `libri` (`title`, `author`, `publication_year`, `price`)
VALUES ("Lo Hobbit", "Tolkien", 1937, 10.45f);
```

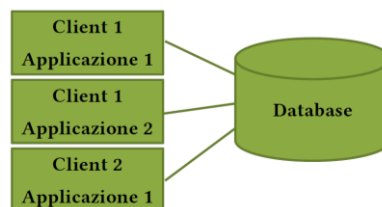
Anche la gestione dell'organizzazione dei dati strutturati è fortemente astratta:

```
ALTER TABLE `libri`
ADD COLUMN `editore` varchar(10)
AFTER `author`;
```

Le applicazioni... sono distribuite sulla rete

I DBMS sono componenti fondamentali di applicazioni complesse e consentono:

- sviluppo di applicazioni differenti che sfruttano la stessa base di dati,
- sviluppo di client differenti per piattaforme differenti che implementano la stessa logica.



Interagire con il DBMS direttamente mediante socket allungherebbe eccessivamente i tempi di sviluppo. A tal proposito, i DBMS relazionali offrono dei protocolli di connessione e trasferimento dati *orientati alle tabelle*: ogni operazione remota restituisce uno o più *result set* in forma tabellare. Tipicamente, i protocolli sono diversi per implementazioni diverse di DBMS (a volte cambiano anche in funzione della versione).

Esistono librerie applicative sia per *mediare* la connessione, sia per *memorizzare* i dati in appositi buffer di memoria. Di seguito, vediamo due esempi:

- Esempio in Java:

```
Properties connectionProps = new Properties();
connectionProps.put("user", "username");
connectionProps.put("password", "password");
Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/",
                                             connectionProps);
```

- Esempio in C:

```
#include <mysql.h>
...
static MYSQL *conn = mysql_init(NULL);
mysql_real_connect(conn, "localhost", "username", "password", "schema-name",
                  3306, NULL, 0);
```

Livelli di architettura

Quando si progetta un'applicazione web ci sono due necessità fondamentali:

1. persistenza dei dati → DBMS
2. implementazione della logica applicativa → ?

Per la seconda necessità abbiamo due possibili soluzioni architetturali:

- architettura a due livelli (2-tier architecture)
- architettura a tre livelli (3-tier architecture)



Nell'**architettura a due livelli** troviamo

1. Primo livello: *client*
 - a. Fornisce l'interfaccia utente al sistema
 - b. Fornisce servizi di presentazione
 - c. Implementa la logica applicativa (in forma limitata o pari a zero nel caso di *thin client*)
2. Secondo livello: *server dati*
 - a. Implementa la logica applicativa (praticamente completamente, nel caso di *thin client*)
 - b. Fornisce servizi di gestione dei dati

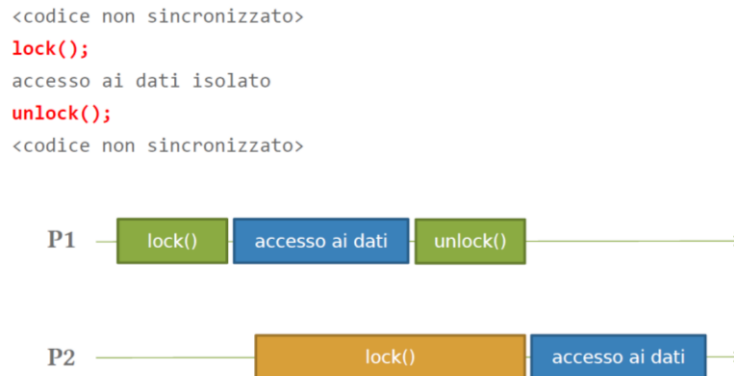
Nell'**architettura a tre livelli** invece troviamo

1. Primo livello: *client*
 - a. Fornisce l'interfaccia utente al sistema
 - b. Fornisce servizi di presentazione
2. Secondo livello: *server applicativo*
 - a. Implementa tutta la logica applicativa
 - b. Si pone come "ponte" tra il client e il server dati

- c. Introdotto per cercare di rendere più efficiente la comunicazione tra il client e il livello dei dati
- 3. Terzo livello: *server dati*
 - a. Fornisce servizi di gestione dei dati

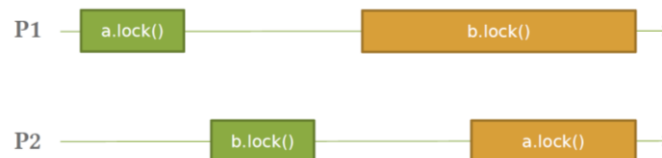
Le applicazioni... sono parallele

Poiché le applicazioni operano spesso in parallelo potrebbe presentarsi il problema della **concorrenza**, risolvibile mediante una sincronizzazione esplicita. Vediamo un esempio.



Problemi legati all'uso dei lock

Nel caso in cui i lock dovessero essere acquisiti nell'ordine sbagliato potrebbe presentarsi un **deadlock**.



I lock non sono componibili! Infatti, non è possibile costruire un programma corretto più grande partendo da piccoli pezzi funzionanti. Il recupero degli errori è complicato: è necessario ripristinare esplicitamente gli invarianti dell'applicazione e rilasciare i blocchi nei gestori delle eccezioni.

Composizione dei lock

```

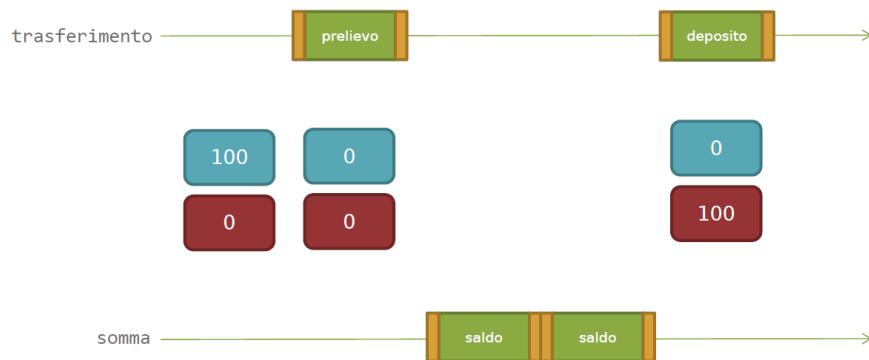
DataStructure ContoBancario {
    Euro importo;
    Lock mutex;
    public deposito(Euro d) {
        mutex.lock();
        importo += d;
        mutex.unlock();
    }
    public prelievo(Euro p) {
        mutex.lock();
        if(importo > p)
            importo -= p;
        mutex.unlock();
    }
    public Euro saldo() {
        return importo;
    }
}

void trasferimento(ContoBancario da, ContoBancario a, Euro valore) {
    da.prelievo(valore);
    a.deposito(valore);
}

Euro somma(ContoBancario c1, ContoBancario c2) {
    return c1.saldo() + c2.saldo();
}

```


Nella foto successiva le parti in arancione rappresentano le operazioni di *lock* e di *unlock*.



Il problema potrebbe essere risolto aggiungendo ulteriori lock, ma non basta inserirli nelle operazioni composte! Inoltre, è un processo che richiede tempo e denaro e, soprattutto, lo stesso problema potrebbe ripresentarsi qualora queste operazioni dovessero venire a loro volta composte.

L'astrazione che ci permette di risolvere il problema una volta per tutte è la **transazione**, ossia una unità atomica di elaborazione.

```
void trasferimento(ContoBancario da, ContoBancario a, Euro valore) {
    start transaction;
    da.prelievo(valore);
    a.deposito(valore);
    commit transaction;
}

Euros sum(Account a1, Account a2) {
    start transaction;
    return a1.read() + a2.read();
    commit transaction;
}
```

Essa non può lasciare la base di dati in uno stato intermedio:

- un guasto o un errore prima del commit debbono causare l'*annullamento* delle operazioni svolte
- un guasto o un malfunzionamento dopo il commit non deve avere conseguenze; se necessario vanno *ripetute* le operazioni.

Vediamo più nel dettaglio l'esito di una transazione:

- **commit**: caso in cui tutto va a buon fine e i dati sono resi *persistenti*
- **abort** (o *rollback*)
 - richiesto dall'applicazione
 - richiesto dal sistema (violazione dei vincoli, concorrenza, incertezza in caso di fallimento).

Interrogazioni al DBMS

L'interazione con l'interprete SQL non è pensata per gli utenti dei sistemi informativi. Tipicamente, vi è un'applicazione che si frappone tra il DBMS e l'utente ed è quindi necessario un supporto per l'interazione programmatica con il DBMS: SQL embedded, Call Level Interface, Object Relational Mapping.

- **SQL embedded** – Lo standard SQL prevede l'esistenza di comandi che possono essere inseriti all'interno di qualsiasi linguaggio di programmazione (`exec sql ...;`). Un compilatore tradizionale non è in grado di interpretare questi comandi ed è quindi necessario l'utilizzo di un preprocessore che trasforma questi comandi in chiamate a libreria per connettersi con uno specifico DBMS da uno specifico sistema. Il preprocessore deve raccordare il codice SQL con il codice nativo (es. per l'accesso alle variabili).
- **Call Level Interface (CLI)** – La *Call Level Interface* (CLI) non è altro che un insieme di librerie scritte pensando ad uno specifico linguaggio di programmazione e sistema. Esistono soluzioni che hanno come obiettivo un solo DBMS o gruppi di DBMS. La CLI espone delle API che possono essere invocate esplicitamente e il risultato di una invocazione viene gestito dalla libreria. La libreria fornirà determinate API per manipolare i risultati delle interrogazioni.

Object Relational Mapping (ORM)

Ogni volta che si effettua un'interrogazione al DBMS, viene creata una *copia* dei dati contenuti nella base di dati. Se si effettua una modifica a questa copia, si crea un *disallineamento* tra i dati conservati in memoria e quelli nella base di dati. Utilizzando gli approcci precedenti, è necessario *esplicitamente* effettuare una sincronizzazione tra le copie (tali operazioni possono rappresentare una percentuale rilevante del codice applicativo da realizzare) e se si modifica lo schema dei dati è necessario aggiornare il codice dell'applicazione (problema di manutenibilità).

Un **sistema di mappatura relazionale degli oggetti** consente di:

- definire una relazione tra tabelle e oggetti (le classi sono la rappresentazione programmatica delle tabelle);
- istanziare automaticamente gli oggetti dal risultato di una interazione con il DBMS;
- rendere persistenti le modifiche agli oggetti.

Tipicamente, un sistema ORM ha necessità di un *generatore di query*.

JDBC

Il Java DB Connectivity Framework è una libreria di integrazione della piattaforma Java che permette di connettersi a DBMS relazioni ed effettuare interrogazioni. Offre driver specifici per interagire con più DBMS, permette uno sviluppo agnostico rispetto alla scelta del DBMS ed offre astrazioni mediante una API comune in `java.sql.*`.

Vari driver JDBC possono essere utilizzati con la stessa interfaccia:

1. JDBC-ODBC (Open DB Connectivity)
 - Uno standard legacy per la connessione con i DBMS
 - Le invocazioni JDBC sono mappate su ODBC
2. Driver Nativi (punto in cui si connette JDBC con quello che vedremo dopo)
 - Le invocazioni a JDBC sono mappate su chiamate CLI di una libreria nativa
 - Occorre avere l'installazione corretta della libreria nativa per il sistema
3. Network Protocol Driver
 - Scritti direttamente in Java e ospitabili su macchine remote
 - Il driver si comporta da middleware (applicazione esterna che si pone nel mezzo tra due applicativi) rispetto alla connessione con il DBMS
4. Altri driver
 - Scritti direttamente in Java e integrati localmente all'applicazione (reinterpretazione in Java della CLI).

java.sql.Connection

Questa classe rappresenta una sessione di comunicazione con il DBMS: la **connessione** deve essere instaurata prima di poter effettuare qualsiasi interrogazione al DBMS. Su questa connessione, occorre specificare l'endpoint di connessione, il nome dello schema, una coppia username/password con privilegi sufficienti per accedere allo schema. È buona pratica aprire poche connessioni e mantenerle attive quanto più a lungo possibile.

```
Properties connectionProps = new Properties();
connectionProps.put("user", "username");
connectionProps.put("password", "password");
DriverManager.getConnection("jdbc:mysql://localhost:3306/schema",
                             connectionProps);
```

java.sql.Statement

Questa classe astrae il concetto di operazione sul DB. Uno **statement** è un comando che viene inviato all'interprete SQL e che deve essere associato ad una connessione attiva

```
connection.createStatement()
```

java.sql.ResultSet

Questa classe astrae il concetto di insieme di tabelle restituite da un'interrogazione. È possibile scandire le tabelle: data una tabella è possibile scandire le singole righe e data una riga è possibile accedere alle singole colonne o utilizzando un indice, o utilizzando il nome della colonna.

La conversione a tipi Java può essere richiesta alla libreria JDBC.

Data Access Object (DAO) Pattern

Il pattern **Data Access Object (DAO)** è un *pattern strutturale*: permette di isolare il livello applicativo/business dal livello di persistenza utilizzando un'API astratta (nasconde all'applicazione la complessità dell'esecuzione di operazioni CRUD – Create, Read, Update, Delete – nel meccanismo di archiviazione sottostante).

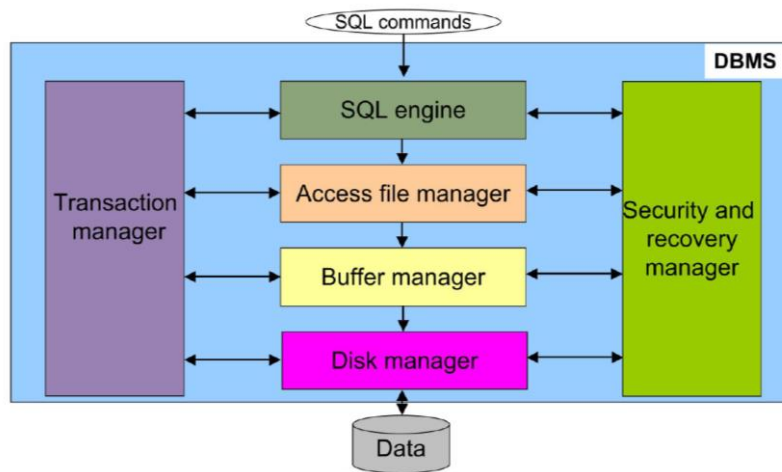
Questo pattern permette ad entrambi i livelli di evolvere separatamente senza sapere nulla l'uno dell'altro. Tipicamente, si utilizzano una **classe di dominio** (che descrive il mini-mondo di riferimento) e un *DAO* in maniera congiunta: la classe di dominio è sotto il

controllo della logica applicativa, mentre il DAO si comporta da ponte tra la logica applicativa ed il livello di persistenza. Inoltre, occorre prestare particolare attenzione ai *vincoli di integrità referenziale*.

Organizzazione fisica

Il DMBS è composto da diversi moduli:

- *gestore transazioni*, che interagisce con tutti i vari moduli per garantire la coerenza delle scritture sui dati;
- *SQL engine*, che riceve le stringhe e crea l'execution plan;
- *gestore accesso ai file*, che recupera dati dal disco;
- *buffer manager*, una porzione del sistema che si preoccupa di gestire una cache in memoria dei dati recuperati da disco;
- *gestore disco*, che andrà ad implementare differenti strategie di accesso ai dati e di recupero degli stessi.



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

I dispositivi di memoria secondaria sono organizzati in *blocchi* di lunghezza (di solito) *fissa* (ordine di grandezza: alcuni KB). Le uniche operazioni sui dispositivi sono la lettura e la scrittura di una *pagina*, cioè dei dati di un blocco (cioè di una stringa di byte). Per comodità consideriamo *blocco* e *pagina* sinonimi.

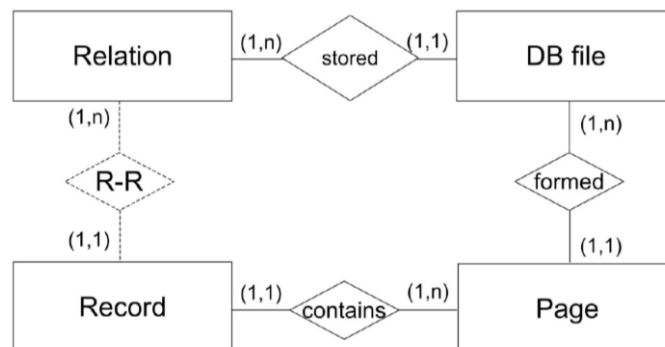
DBMS e file system

Il file system è il componente del sistema operativo che gestisce la memoria secondaria. I DBMS ne utilizzano le funzionalità, ma in misura limitata, per creare file e per leggere e scrivere singoli blocchi o sequenze di blocchi contigui.

L'organizzazione dei file, sia in termini di distribuzione dei record nei blocchi, sia relativamente alla struttura all'interno dei singoli blocchi è gestita direttamente dal DBMS. Quest'ultimo gestisce i blocchi dei file allocati come se fossero un unico grande spazio di memoria secondaria e costruisce, in tale spazio, le strutture fisiche con cui implementa le relazioni.

Pagine e record

Con il termine **record** indichiamo il nome di una tupla da parte del DBMS (è una rappresentazione in forma strutturata della memoria che l'applicativo può gestire).



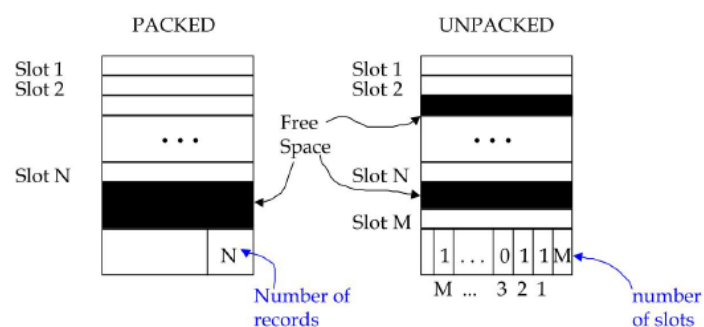
Dallo schema possiamo dire che dato un record possiamo sapere a quale relazione esso appartiene.

La dimensione tipica di una pagina su cui lavora il DBMS è quella di un blocco, il quale contiene un insieme di record (tipicamente di una relazione). Fisicamente, è una collezione di **slot**, ovvero un insieme di byte, (spazio che manteniamo per una tupla all'interno di una pagina) e ne abbiamo uno per ciascun record. Ciascun record ha un identificatore detto **record id** (o *rid*), il quale ha la seguente forma: $rid = \langle page\ id, slot\ number \rangle$.

Pagine con record di lunghezza fissa e variabile

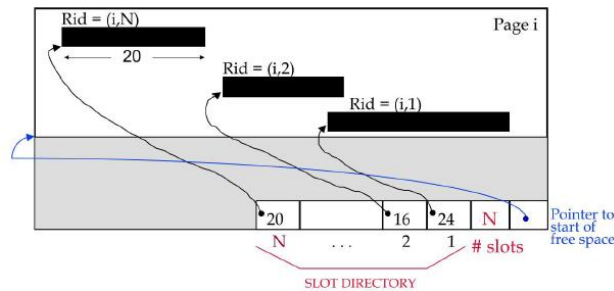
Le pagine possono avere sia record di taglia fissa, sia record di taglia variabile. Nel primo caso possono avere un'organizzazione:

- **impacchettata**: il DBMS già sa di quanti byte ha bisogno per memorizzare una relazione, in quanto prende il blocco, lo spaccetta e crea tanti slot quanti sono i record che possono entrare nella pagina. Tutti i record vengono mantenuti all'inizio e lo spazio libero alla fine. In caso di eliminazione di uno slot occorre spostare tutti i record; quindi, fornisce prestazioni migliori per inserimento e ricerca, ma peggiori in caso di eliminazione. Inoltre, lo spostamento di un record cambia la sua posizione, e questo è un problema quando i record sono citati da altre pagine.
- **non impacchettata**: lo spazio libero è sparso all'interno della pagina, garantendo una diversa gestione delle eliminazioni. L'identificazione di un record richiede la scansione della bitmap per verificare se lo slot è libero.



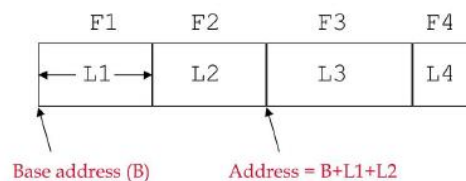
Nel caso in cui le pagine abbiano invece record di taglia variabile, possiamo utilizzare altri tipi di organizzazione, in cui per sapere dove si trova un record abbiamo bisogno del suo indirizzo e della sua lunghezza. Viene creata una directory da N slot, in cui manteniamo i puntatori ai record e un puntatore all'inizio dello spazio libero; inoltre, manteniamo anche l'informazione riguardo la dimensione di ciascuno slot.

Qui non avremmo nessun problema nello spostare i record. Cancellare un record significa impostare a -1 il valore dello slot corrispondente e spostare lo spazio del record nello spazio libero (area dati e spazio libero riorganizzati quando necessario).



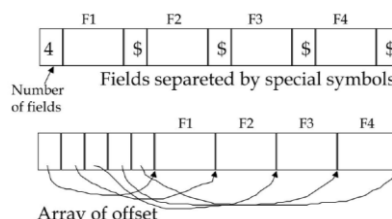
Formato di un record a taglia fissa e variabile

Nei record a taglia fissa le informazioni sui campi sono le stesse per tutti i record di un certo tipo e sono memorizzate nel *catalogo del sistema*. Per trovare il campo i -esimo è necessario scandire il record.



Nei record a taglia variabile bisogna mantenere (all'interno del record) dei metadati per conoscere la posizione e la dimensione dei campi. La quantità di byte che servono per rappresentare i metadati è nota. Vi sono due alternative di rappresentazione:

- **terminatore di campo:** c'è un metadato iniziale che ci dice quanti sono i campi all'interno del record e ogni volta che troviamo il terminatore questo ci indica che un campo è finito;
- **vettore di spiazamento:** all'interno dello slot troviamo un valore numerico che ci dice dopo quanti byte troveremo un campo.



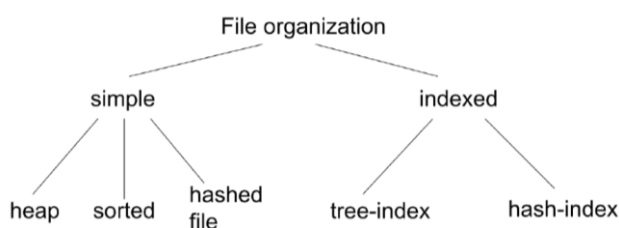
La seconda alternativa consente un accesso diretto ai campi ed una memorizzazione efficiente dei valori nulli (due puntatori consecutivi identici identificano un valore nullo).

Organizzazione dei file

Quindi, abbiamo visto che un file è una collezione di pagine, ciascuna contenente una collezione di record. Sui file è necessario supportare le seguenti operazioni:

- inserimento/eliminazione/aggiornamento di un record;
- lettura di un record dato il suo rid;
- scansione di tutti i record, eventualmente concentrandosi sui record che soddisfano una data condizione.

È possibile organizzare i file secondo due tipi di organizzazione: semplice e indicizzata.



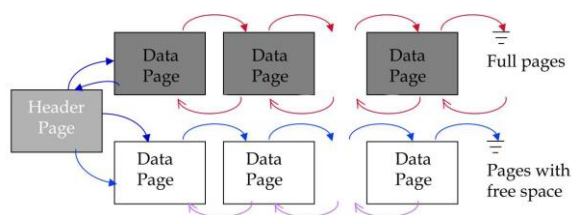
Heap file

Nell'**organizzazione a mucchio**, il file che rappresenta la relazione contiene il record nelle pagine, senza alcun criterio particolare. Quando la relazione cresce o si riduce, le pagine vengono allocate o deallocate. Per supportare le operazioni, è necessario tenere traccia:

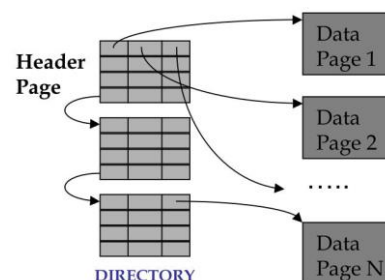
- delle pagine appartenenti al file;
- dello spazio libero nelle pagine del file;
- dei record nelle pagine del file.

La distinzione tra le pagine piene e quelle non piene viene effettuata o tramite delle liste, o tramite una directory.

Nella *rappresentazione tramite liste* avremo due liste separate, una per le pagine piene ed un'altra per quelle che hanno dello spazio libero. Quando è necessaria una pagina, la richiesta viene inoltrata al *gestore del disco*: la pagina restituita dal gestore viene inserita come prima pagina dell'elenco delle pagine con spazio libero. Quando una pagina non è più utilizzata (cioè è costituita solo da spazio libero), viene cancellata dall'elenco delle pagine con spazio libero.



Nella *rappresentazione con directory* si utilizza, appunto, una *directory*, ossia un elenco di pagine in cui ogni entry contiene un puntatore ad una pagina dati. Ogni entry si riferisce a una pagina e indica se la pagina ha spazio libero (attraverso un contatore). La ricerca di una pagina con spazio libero è più efficiente, in quanto il numero di accessi è lineare rispetto alla dimensione della directory, non alla dimensione della relazione.



File con pagine ordinate

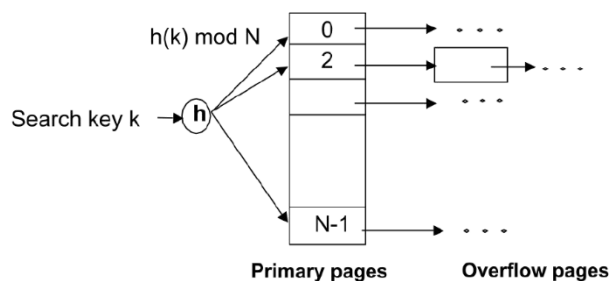
I record sono ordinati secondo un *insieme di campi* (la **chiave di ricerca**). Le pagine sono ordinate secondo l'ordinamento dei propri record e sono organizzate in una struttura sequenziale (l'ordine delle pagine riflette l'ordinamento dei record). Questa organizzazione generale può essere implementata con strutture dati differenti.

Hashed file

Nei **file con hash** le pagine della relazione sono organizzate in gruppi (**bucket**). Un bucket consiste di:

- una pagina *primaria*;
- possibili altre pagine (di *overflow*) collegate alla pagina primaria.

Un insieme di campi della relazione è scelto come chiave di ricerca: per cercare un record R , data una chiave di ricerca K , si può individuare l'indirizzo del bucket che contiene R usando una *funzione hash*.



Modello di costo (tempo di esecuzione)

Le operazioni di I/O dominano il processamento in memoria centrale. Per caratterizzare il tempo di esecuzione delle operazioni, ci possiamo concentrare sul numero di pagine accedute. I parametri del modello di costo sono:

- B – numero di pagine nel file;
- R – numero di record per pagina;
- D – tempo medio di scrittura/lettura di una pagina (es. 15ms);
- C – tempo medio di processamento di un record (es. 100ns), che considera anche aspetti come il confronto di un campo con un valore.

Le **operazioni sui dati** che dovremmo andare a supportare sono:

- *Scansione* dei record di un file e pagheremo
 - costo di caricamento delle pagine del file
 - costo di CPU per individuare i record nelle pagine
- Selezione basata su *uguaglianza* e pagheremo
 - costo di caricamento delle pagine con record rilevanti
 - costo di CPU per identificare i record nelle pagine
 - il record da individuare è unico se l'uguaglianza è sulla chiave
- Selezione basata su *intervallo* e pagheremo
 - costo di caricamento delle pagine con record rilevanti

- costo di CPU per identificare i record nelle pagine
- *Inserimento* di un record e pagheremo
 - costo di individuazione della pagina in cui avviene l'inserimento
 - costo di caricamento della pagina
 - costo di modifica della pagina
 - costo di riscrittura della pagina modificata
 - costo di caricamento, modifica e scrittura di *altre* pagine se necessario
- *Eliminazione* di un record e pagheremo
 - le stesse operazioni dell'inserimento

Istanziamento del modello per heap file

Il costo della scansione sarà $B(D + RC)$. Infatti, per ciascuna delle B pagine del file:

- carica (D)
- per ciascuno degli R record della pagina: processalo (C).

Il costo della selezione basata su uguaglianza sarà $B(D + RC)$, in quanto il record può non essere presente o più di un record può soddisfare la condizione (e in entrambi i casi andrebbero letti tutti i record). Lo stesso costo lo abbiamo per la selezione basata su intervallo.

Il costo dell'inserimento sarà $D + C + D$, perché (nell'ordine):

- caricamento dell'ultima pagina
- inserimento nella pagina
- scrittura della pagina

Il costo della eliminazione:

- se il record è identificato dal rid $\rightarrow D + C + D$
- se il record è identificato da una selezione di uguaglianza o su intervallo $\rightarrow B(D + RC) + XC + YD$
 - X è il numero di record da eliminare
 - Y è il numero delle pagine contenenti record da eliminare

Istanziamento del modello per file ordinato

Il costo della scansione sarà $B(D + RC)$.

Il costo della selezione basata su uguaglianza sarà $D \log_2 B + C \log_2 R$ (caso peggiore):

- ricerca binaria per identificare la pagina con il primo record rilevante
- $\log_2 B$ passi per identificare la pagina
- ad ogni passo, un'operazione di I/O è due confronti (che possiamo ignorare)
- ricerca binaria per identificare il primo record rilevante nella pagina ($\log_2 R$)

Il costo della selezione basata su intervallo sarà simile al caso precedente, ma avrà un costo maggiore per il caricamento di ulteriori pagine, se i record rilevanti non sono memorizzati in una singola pagina.

Il costo dell'inserimento sarà $D \log_2 B + C \log_2 R + C + 2B(D + RC)$:

- (caso peggiore, in cui dovremmo rispostare tutto) prima pagina, prima posizione
- costo di ricerca per la pagina in cui effettuare l'inserimento: $D \log_2 B + C \log_2 R$
- inserimento del record: C
- caricamento e scrittura delle altre pagine: $2B(D + RC)$

Il costo dell'eliminazione sarà simile al caso precedente. Se la condizione di eliminazione è una selezione di uguaglianze su campi non chiave, o una selezione per intervallo, il costo dipende anche dal numero di record da eliminare.

Istanziamento del modello per file hash (approssimata)

Il costo della scansione sarà $1.25B(D + RC)$: assumiamo (come è tipico) che le pagine hanno un'occupazione pari all'80% per minimizzare gli overflow quando il file cresce.

Il costo della selezione basata su uguaglianza sarà $(D + RC) \cdot \text{numero di record rilevanti}$ (assumiamo accesso diretto attraverso la funzione hash). Il costo della selezione basata su intervallo invece sarà $1.25B(D + RC)$.

Il costo dell'inserimento sarà $2D + RC$.

Il costo dell'eliminazione sarà *costo della ricerca* + $D + RC$.

File indicizzati

Un **indice** è una struttura dati ausiliaria che permette l'ottimizzazione della ricerca di un record in un file, basata sul valore di uno o più campi prefissati, che formano la cosiddetta *chiave di ricerca*. Un qualsiasi sottoinsieme dei campi di una relazione può formare la chiave di ricerca per l'indice.

Nota: la *chiave di ricerca* è differente dalla *chiave* di una relazione. La chiave di ricerca può essere l'insieme minimo di attributi che identificano i record di una relazione.

L'implementazione di una relazione R mediante un'organizzazione basata su indici comprende vari elementi. Abbiamo un **file di indice**, contenente:

- *entry dati*, ciascuna contenente un valore K della chiave di ricerca e che vengono utilizzate per identificare nel file dati i record dati associati al valore K della chiave di ricerca
- *entry indice*, utilizzate per la gestione del file di indice (non necessariamente presente in tutte le organizzazioni)

ed abbiamo un **file di dati**, contenente i record nella relazione R .

L'organizzazione del file di indice può seguire differenti strategie: può essere basato su alberi oppure su funzioni hash. Un indice:

- è caratterizzato dalle strutture delle entry dati;
- può essere raggruppato/non raggruppato;
- può essere denso/sparso;
- può essere primario/secondario;
- può presentare chiavi semplici o composite.

Struttura di una entry dati

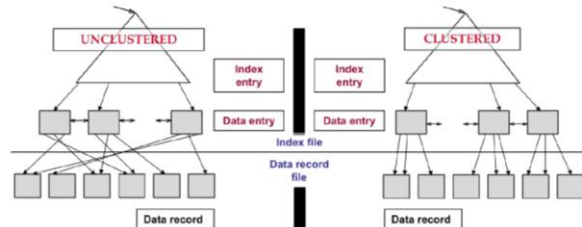
Vi sono tre tecniche *alternative* per memorizzare una entry dati (K^*) la cui chiave di ricerca è K :

1. K^* è un record dati: caso estremo, perché le entry dati non sono separate dai record di dati (caso del file hash);
2. K^* è una coppia (K, rid) : il file di indice è indipendente dal file dati;
3. K^* è una coppia $(K, lista - rid)$
 - a. lista-rid è una lista di identificatori di record dati con chiave di ricerca pari a K ;
 - b. il file di indice è indipendente dal file dati;
 - c. utilizzo migliore dello spazio, al prezzo dell'utilizzo di entry dati di taglia variabile.

Se utilizziamo più di un indice sullo stesso file dati, al più uno può utilizzare la prima tecnica.

Raggruppato/non raggruppato

Un indice è **raggruppato** quando le sue entry dati sono memorizzate secondo un ordine coerente con (o identico a) l'ordine dei record dati nel file dati. In caso contrario, l'indice è **non raggruppato**.



Un indice le cui entry sono memorizzate con la tecnica 1 è raggruppato per definizione (infatti l'ultimo livello sarà proprio il data record). Nel caso delle alternative, un indice è raggruppato solo se i record dati sono memorizzati nel file dati secondo la chiave di ricerca.

Se l'indice non è raggruppato, *non può essere usato* per le ricerche basate su intervallo, in quanto non possiamo sapere quali sono i valori che effettivamente ricadono nell'intervallo visto che ne viene letto uno solo. Ci può essere al più un indice raggruppato per ciascun data file, poiché l'ordine dei record dati nel file dati può essere coerente con al più una chiave di ricerca associata ad un solo indice.

Indici primari e secondari

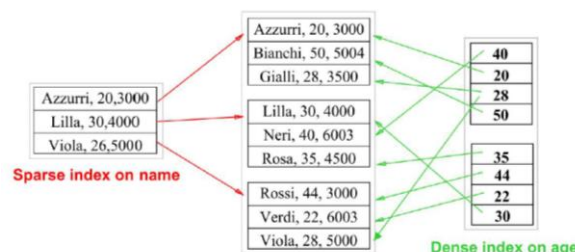
Un **indice primario** è un indice di una relazione R la cui chiave di ricerca include la chiave primaria di R . Se un indice non è primario, allora è **secondario**.

Consideriamo due entry dati *duplicate* (ossia, con lo stesso valore della chiave di ricerca):

- un indice primario non può contenere duplicati;
- un indice secondario contiene tipicamente duplicati;
- un indice secondario è chiamato *unico* (unique) se la sua chiave di ricerca contiene una chiave (non primaria). Un indice secondario unico non contiene duplicati. Può essere usato per garantire l'unicità di attributi che comunque non costituiscono la chiave primaria.

Indici sparsi e densi

Un indice è **denso** se ogni valore della chiave di ricerca che compare nel file dati compare anche in almeno una entry dati dell'indice. In caso contrario, l'indice è **sperso**.



Un indice che usa la tecnica 1 è denso per definizione. Un indice sparso è più compatto ed è raggruppato (vi è un indice sparso per ciascun file dati). Se un indice ha più riferimenti, allora viene usata o la tecnica 2 o la tecnica 3.

Chiavi semplici o composite

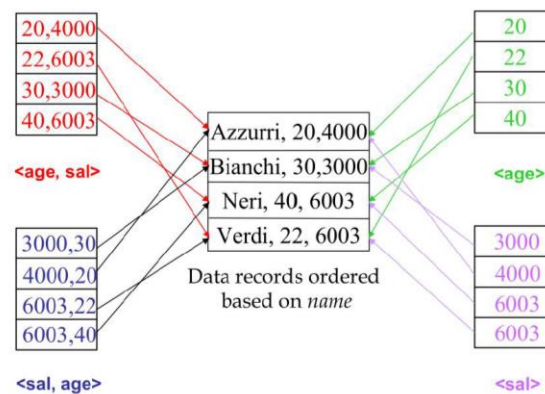
Una chiave di ricerca è **semplice** se è composta da un singolo attributo. Altrimenti, è **composita** (richiede di sapere com'è fatto l'indice (?)). Se la chiave è composita:

- una query basata su uguaglianza (*equality query*) è una query che ha fissato ogni valore degli attributi
- altrimenti, è una query su intervallo (*range query*).

Un indice composito supporta un gran numero di query:

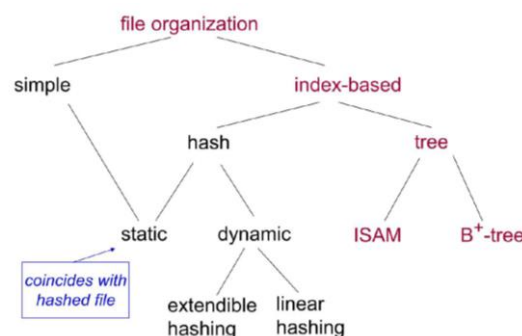
- una singola query può estrarre più informazioni;
- possiamo anche evitare di accedere al data record se i campi rilevanti della query sono parte della chiave di ricerca.

Inoltre, un indice composito viene aggiornato più frequentemente (ed ha quindi un costo di gestione maggiore). Di seguito, un esempio con chiave semplici e composite:



Indici e organizzazioni degli indici

Possiamo avere un'organizzazione dei file basata su indice, che sfrutta due tecniche principali: hash (statica, ovvero l'indice è un file hash, o dinamica) e alberi (ISAM o B+).



Indice ad albero

Esempio di interrogazione: *trova tutti gli studenti con media > 27*. Se gli studenti sono ordinati per media, possiamo usare la ricerca binaria; tuttavia, la ricerca binaria può essere costosa per file grandi.

Un possibile approccio di ottimizzazione consiste nell'usare una struttura dati ausiliaria (l'indice) che riferisce il file ordinato. In questo modo, la ricerca binaria può essere eseguita

su un file più piccolo: le entry dell'indice sono più piccole e ci sono meno entry (una per pagina del file dati).



Se iteriamo con questo processo fino a quando la struttura ausiliaria entra in una sola pagina, otteniamo un **indice ad albero**.

In un indice ad albero le entry dati sono organizzate in una struttura ad albero, basata sul valore della chiave di ricerca. Cercare un record significa cercare la pagina con l'entry dati desiderata con l'ausilio dell'albero:

- ogni nodo descrive una pagina;
- le pagine con le entry dati sono le foglie dell'albero;
- le ricerche partono dalla radice e terminano su una foglia;
- i collegamenti tra i nodi corrispondono a puntatori tra pagine.

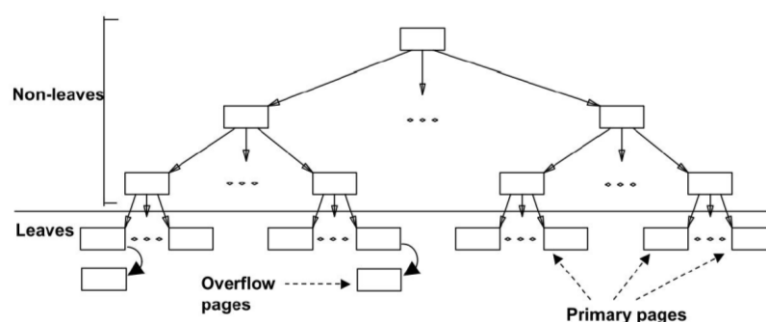
La struttura di un nodo intermedio e della radice è la seguente:



- Sequenza di $m + 1$ puntatori P , separati da vari valori K ordinati secondo la chiave di ricerca.
- Il puntatore P_{i-1} a sinistra del valore K_i ($1 \leq i \leq m$) punta al sottoalbero contenente entry dati con valori minori di K_i .
- Il puntatore P_{i+1} a destra del valore K_i ($1 \leq i \leq m$) punta al sottoalbero contenente entry dati con valori maggiori di K_i .

Indexed Sequential Access Method (ISAM)

L'**ISAM** (*Indexed Sequential Access Method*) è un modo per immagazzinare dati da estrarre rapidamente. Le foglie contengono le entry dati e possono essere scandite sequenzialmente. La struttura dell'indice è *statica* (quando popoliamo il DB viene costruito l'indice, che tende a non cambiare), soggetta a costose riorganizzazioni periodiche. Una volta arrivati alle foglie, possiamo trovare delle pagine di overflow (che potrebbero scomparire in seguito a dei ribilanciamenti).

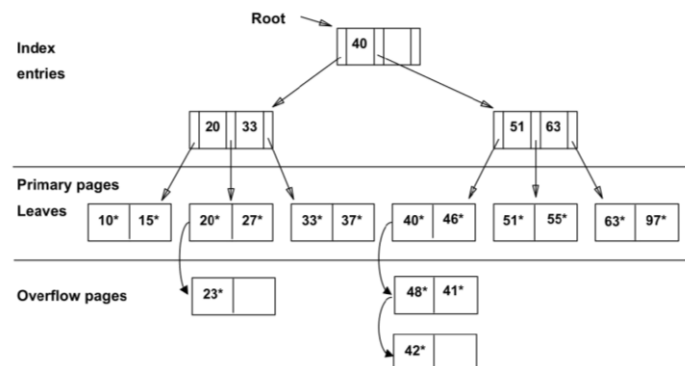


Un ISAM è un **albero bilanciato**: il percorso dalla radice alla foglia ha la stessa lunghezza per tutte le foglie. L'*altezza* di un albero bilanciato è la lunghezza del percorso dalla radice alla foglia. In ISAM, ogni nodo non foglia ha lo stesso numero di figli (*fan-out* dell'albero, che ci dice quanti figli può avere un determinato nodo). Se ogni nodo ha F figli, un albero di altezza h ha F^h pagine di foglie.

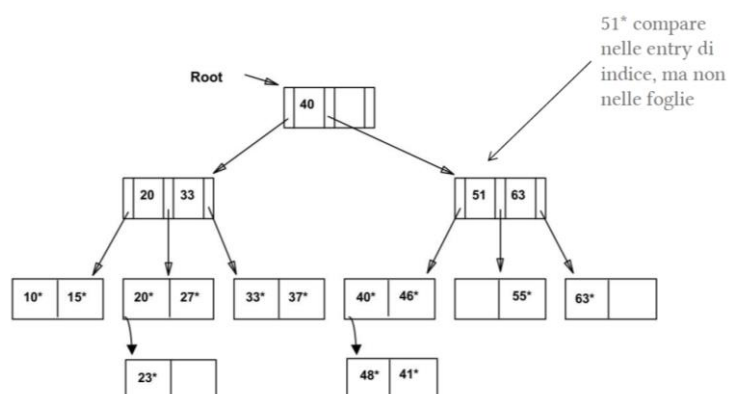
Tipicamente, il fan-out F è almeno 100, per cui un albero di altezza 4 contiene 100 milioni (100^4) di pagine foglia. Questo implica che possiamo trovare la pagina desiderata utilizzando 4 operazioni di I/O (o 3, se la radice è nel buffer) e che una ricerca binaria dello stesso file richiederebbe $\log_2 100.000.000 (> 25)$ operazioni di I/O.

- *Creazione del file*: le foglie sono allocate in modo sequenziale.
- *Ricerca*: partiamo dalla radice e confrontiamo la chiave che stiamo cercando con le chiavi dell'albero, finché non si arriva a una foglia (costo $\log_F N$). In genere, $N = B/F$, dove B è il numero di pagine del file di dati.
- *Inserimento*: troviamo la foglia corretta in cui inserire, allocando una pagina di overflow se necessario, e poi si inserisce il record di dati nel file di dati.
- *Eliminazione*: troviamo e cancelliamo dalle foglie. Se la pagina è una pagina di overflow ed è vuota, la deallochiamo; in ogni caso, cancelliamo il record di dati corretto dal file di dati.

Vediamo un esempio, in cui assumiamo che ciascun nodo contenga 2 entry. Supponiamo di voler inserire 23^* , 48^* , 41^* , 42^* :



Se poi volessimo eliminare 42^* , 51^* , 97^* avremmo:



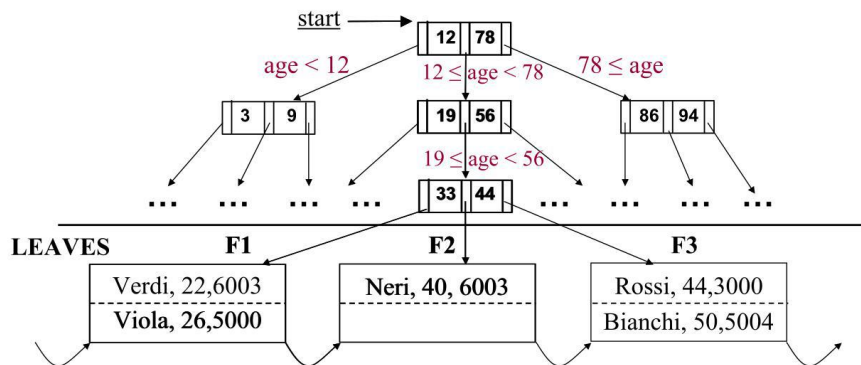
Notiamo come la chiave 51 rimanga lì, nonostante non ci sia più nessun record con valore 51^* : rimarrà lì finché non verrà aggiornato.

Indice B⁺-tree

Un **B⁺-tree** è un albero *bilanciato* in cui la lunghezza del percorso dalla radice alla foglia è la stessa per tutte le foglie. Questi alberi superano le limitazioni che ISAM ha con l'inserimento e la cancellazione.

Ogni nodo contiene m tuple, con $d \leq m \leq 2d$, dove d è il **rango dell'albero**. Per ogni nodo N (eccetto la radice), m è scelto in modo tale che almeno il 50% di N sia riempito. Le foglie (data entry) sono collegate attraverso una lista ordinata per chiave di ricerca; questo meccanismo è utile per le *range query*: cerchiamo il primo valore dell'intervallo e accediamo alla foglia L “corretta”, dopodiché scorriamo la lista dalla foglia L alla foglia con l'ultimo valore dell'intervallo.

Vediamo la **ricerca** in un B⁺-tree. Cerchiamo le entry dati con $22 < età \leq 44$. Per prima cosa individuiamo la foglia con il bound minimo nell'albero \rightarrow troviamo $F1 \rightarrow$ scandiamo la lista da $F1$ a $F3$, che contiene il primo record con il valore fuori dall'intervallo.



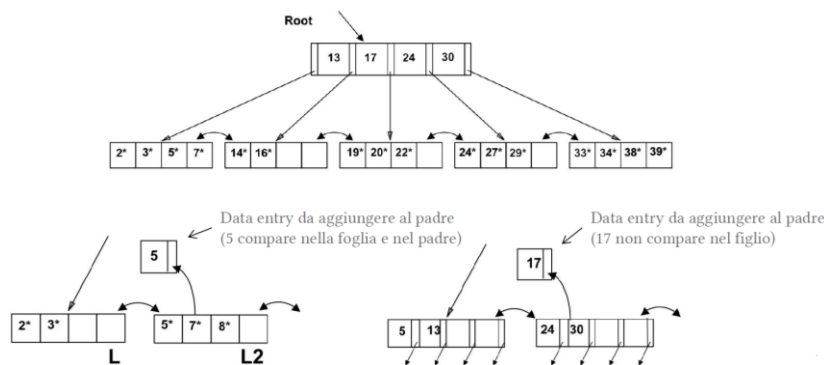
In un B⁺-tree il numero di accessi a pagina necessario per un'operazione di ricerca è al più l'altezza dell'albero: $\log_F N$, dove F è il *fanout medio* ed N è il numero delle foglie. Un valore tipico è $F = 100$: con questo valore, se abbiamo 1 milione di pagine, il costo della ricerca è 4 operazioni di I/O (o 3 se la radice è nel buffer); inoltre, la maggior parte delle pagine è occupata dalle foglie.

I B⁺-tree sono ottimali per le ricerche basate su intervalli, ma si comportano bene anche per la ricerca basata su uguaglianza.

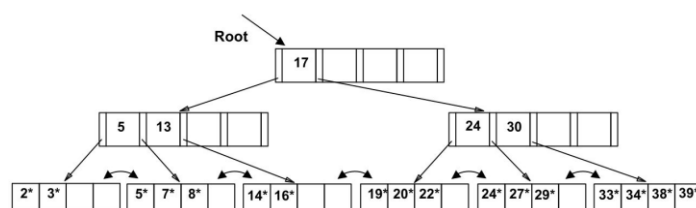
Per mantenere l'albero bilanciato, quando si effettua un *inserimento* in una foglia L piena:

- le tuple di L vengono distribuite equamente in L ed in nuovo nodo $L2$;
- nel genitore di L si inserisce una nuova entry dati che punta a $L2$ (la chiave di ricerca è il minimo tra i valori che compaiono in $L2$);
- ricorsivamente si applica la stessa strategia fino alla radice;
- quando un nodo intermedio viene diviso, spostiamo il valor medio del nodo nel genitore.

Vediamo un esempio di inserimento di un record dati con chiave di ricerca 8:



Con questo inserimento l'albero è cresciuto di altezza:



Tipicamente, l'albero cresce in *ampiezza*. L'unico caso in cui l'albero cresce in altezza è quando effettuiamo un inserimento nella radice piena.

Assumiamo l'alternativa 1 (indice B⁺-tree raggruppato). Studi empirici dimostrano che in media un B⁺-tree ha le foglie piene al 67%: il numero di pagine con entry dati è circa $1.5B$, dove B è il numero minimo di pagine necessario per la memorizzazione delle entry dati. Il numero di pagine fisiche per le foglie è $B' = 1.5B$.

- *Scansione*: $1.5B(D + RC)$
- *Selezione per uguaglianza*: $D \log_F(1.5B) + C \log_2 R$
 - cerca la prima pagina con un record dati di interesse
 - cerca il primo record dati con ricerca binaria nella pagina
 - tipicamente, i record di interesse compaiono in una sola pagina.
- Nella pratica, la radice è bufferizzata, quindi evitiamo un accesso
- *Selezione per intervallo*: come la ricerca per uguaglianza. Operazioni di I/O aggiuntive se i record dati sono inseriti su più foglie collegate
- *Inserimento*: $D \log_F(1.5B) + C \log_2 R + C + D$
 - costo della ricerca + inserimento + scrittura
 - ignoriamo i costi addizionali dovuti all'inserimento in una pagina piena
- *Eliminazione*: analogo all'inserimento. Ignoriamo i costi addizionali dovuti al rilascio di una pagina vuota.

Assumiamo l'alternativa 2 (indice B⁺-tree non raggruppato) e assumiamo che il file dati sia un heap file. Supponiamo che la dimensione di un'entry dati in una foglia sia $1/10$ di un record dati:

→ numero di foglie nell'indice: $0.1(1.5B) = 0.15B$;

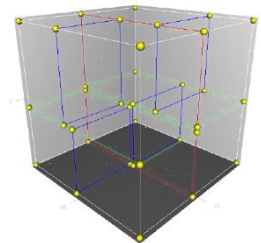
→ numero di entry dati in una pagina foglia (piena al 67%): $10(0.67R) = 6.7R$

- *Scansione*: $0.15B(D + 6.7RC) + BR(D + C)$
 - scansione di tutte le entry dati dell'indice: $0.15B(D + 6.7RC)$
 - ogni data entry in una foglia può puntare a una pagina nel file dati differente: $BR(D + C)$
- Poiché il costo della scansione è altissimo, si usa una strategia alternativa: scandire il file dati e ordinarlo per chiave:
 - file dati composto da B pagine
 - algoritmo a due passaggi: ad ogni passaggio leggi e scrivi l'intero file
 - costo: $4B$, significativamente minore del costo della strategia precedente
- Conviene ignorare l'indice!

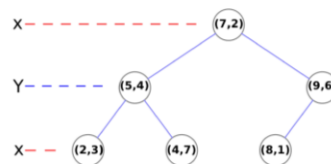
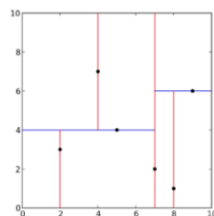
k-d Trees

Le query possono operare su intervalli multidimensionali (e.g. dati spaziali per applicazioni GIS) e si parla in questo caso di **query multidimensionali**. Deve quindi essere possibile indicizzare i dati lungo più assi. I DBMS tipicamente offrono la possibilità di utilizzare indici ottimizzati per questi scopi.

I **k-d Trees** sono una struttura dati utilizzata per partizionare lo spazio e sono molto utilizzati nelle ricerche multidimensionali. Ogni foglia dell'albero è un punto nello spazio, ogni nodo interno “genera” un iperpiano che divide lo spazio in due semispazi: i punti a sinistra del semispazio sono nel sottoalbero sinistro, i punti a destra del semispazio sono nel sottoalbero destro.

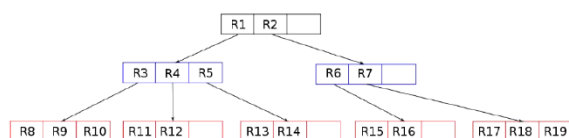
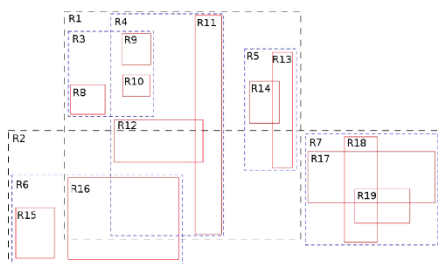


Scendendo di un livello, si genera una partizione nella direzione “successiva”: il primo livello partiziona in x , il secondo livello partiziona in y , il terzo livello partiziona in z , ...



R-Tree

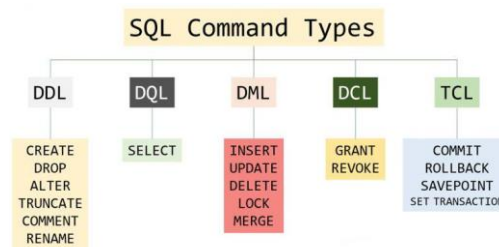
Gli **R-Tree** sono una forma particolare di B-tree che raggruppa oggetti vicini all'interno di **bounding rectangles**. Le foglie descrivono un singolo elemento. Ai livelli superiori, si individuano aggregazioni superiori. I rettangoli possono essere divisi quando il numero di punti supera una certa soglia.



SQL avanzato

L'SQL (*Structured Query Language*) è uno standard che racchiude un insieme di sottolinguaggi:

- **Data Definition Language** (DDL): permette di creare, modificare, eliminare gli oggetti in un database, agendo sullo schema;
- **Data Query Language** (DQL): consente l'estrazione di dati dalla base di dati;
- **Data Manipulation Language** (DML): consente di inserire, modificare, eliminare i dati in un database;
- **Data Control Language** (DCL): per fornire o revocare permessi per utilizzare comandi DML/DDL/DQL ed anche DCL;
- **Transaction Control Language** (TCL): permette di gestire la concorrenza, assicurando la coerenza dei dati.



Vincoli di integrità generici: check e asserzioni

Specifica di vincoli di ennupla o vincoli a livello di logica applicativa:

```
create table Impiegato (
    Matricola integer primary key,
    Cognome character(20),
    Nome character(20),
    Sesso character not null check (sesso in ('M','F')) ,
    Stipendio integer check (Stipendio > 0) ,
    Superiore integer,
    check (Stipendio <= (select Stipendio
                        from Impiegato I
                        where Superiore = I.Matricola) )
);

create assertion NomeAss check ( Condizione )

create assertion `AlmenoUnImpiegato`
    check (1 <= (select count(*)
                from Impiegato ))
```

Questa cosa, presente nello standard di SQL, tuttavia non è supportata. Lo stesso vale per le *asserzioni*, che permettono di implementare vincoli generici (regole aziendali).

Basi di dati attive: i trigger

Una base di dati può contenere regole attive, chiamate **trigger**. Il paradigma generico dei trigger è Evento-Condizione-Evento:

- quando un evento si verifica
- se la condizione è vera
- allora l'azione è eseguita.

La reattività dei trigger sta nel fatto che l'inserimento dei dati in una tabella può causare l'inserimento dello stesso dato in un'altra tabella (e così via per le altre operazioni). Quindi, questo modello consente **computazioni reattive**, ma c'è il rischio di dipendenze circolari tra i trigger.

La sintassi per la creazione dei trigger è la seguente:

```
create trigger NomeTrigger
{ before | after }
{ insert | delete | update [of Column] } on TabellaTarget
[referencing
    {[old table [as] VarTuplaOld]
     [new table [as] VarTuplaNew] } |
    {[old [row] [as] VarTabellaOld]
     [new [row] [as] VarTabellaNew] }]
[for each { row | statement }]
[when Condizione]
StatementProceduraleSQL
```

Le tre classi principali di eventi che possono scatenare un trigger sono l'inserimento, la cancellazione e l'aggiornamento (come vediamo sopra).

Distinguiamo due tipi di eventi:

- **BEFORE**
 - il trigger è considerato e possibilmente eseguito prima dell'evento (i.e. la modifica del database)
 - i trigger before non possono modificare lo stato del database; possono al più condizionare i valori che si stanno per inserire nel database
 - normalmente, questa modalità è usata quando si vuole verificare una modifica prima che essa avvenga e "modificare la modifica"
- **AFTER**
 - il trigger è considerato ed eseguito dopo l'evento

Vediamo un esempio:

Conditioner (agisce prima dell'update e della verifica di integrità)

```
create trigger LimitaAumenti
before update of Salario on Impiegato
for each row
when (New.Salario > Old.Salario * 1.2)
set New.Salario = Old.Salario * 1.2
```

Re-installer (agisce dopo l'update)

```
create trigger LimitaAumenti
after update of Salario on Impiegato
for each row
when (New.Salario > Old.Salario * 1.2)
set New.Salario = Old.Salario * 1.2
```

Distinguiamo poi (la granularità degli eventi):

- modalità **statement-level** (**for each statement**)
 - il trigger viene considerato e possibilmente eseguito solo una volta per ogni comando che lo ha attivato, indipendentemente dal numero di tuple modificate
 - in linea con SQL (set-oriented)
- modalità **row-level** (opzione **for each row**)
 - il trigger viene considerato e possibilmente eseguito una volta per ogni tupla modificata
 - scrivere trigger row-level è più semplice

Le *variabili* e le *tabelle di transizione* dipendono dalla granularità:

- se la modalità è row-level, ci sono due **variabili di transizione** (**old** and **new**) che rappresentano il valore precedente o successivo alla modifica di una tupla;
- se la modalità è statement-level, ci sono due **tabelle di transizione** (**old table** and **new table**) che contengono i valori precedenti e successivi delle tuple modificate dallo statement.

C'è da sottolineare che **old** e **old table** non sono presenti con l'evento **insert**, così come **new** e **new table** non sono presenti con l'evento **delete**.

MySQL non supporta le asserzioni (in particolare il comando **STOP ACTION** non è supportato), ma si possono utilizzare trigger “before update” per emularne il funzionamento. Supponiamo ad esempio di non voler avere più di due elementi per una certa classe di entità:

```
create assertion AT_MOST_TWO as CHECK
((select count(*) from `entity` E
  where E.class = 'THE_CLASS') <= 2
)

create trigger AT_MOST_TWO
before insert on `entity` for each row
begin
  declare counter INT;
  select count(*) from `entity` E
  where E.class = 'THE_CLASS' into counter;
  if counter > 2 then
    signal sqlstate '45000';
  end if;
end
```

Con “into counter” stiamo inserendo il risultato della **select** all'interno della variabile **counter**, precedentemente dichiarata con “declare counter INT”. Con l'if stiamo dichiarando che se il contatore ha un valore maggiore, allora il DBMS deve interrompere l'esecuzione dell'execution plan.

SQL States

Alcune procedure SQL hanno la necessità di fornire al DBMS (che eventualmente lo propagerà al client chiamante) un codice che fornisca delle informazioni sul successo o sul fallimento dell'esecuzione – una sorta di valore di ritorno.

Gli **SQL States** sono codici di 5 byte (i primi due per la classe, i restanti tre per differenziare i codici di errore). Ad esempio:

- 3F000: invalid schema name;
- 30000: invalid SQL statement identifier;
- 23000: integrity constraint violation;
- 22012: data exception – division by zero;
- 45000: unhandled user-defined exception (la classe 45 indica che c'è qualche problema nella logica applicativa che abbiamo implementato).

Creazione di indici

Possiamo anche utilizzare REPLACE per specificare che, se è già stato creato, vogliamo rimpiazzare l'indice.

Possiamo definire il tipo dell'indice (*unique*, *fulltext*, per effettuare ricerche testuali in campi arbitrariamente lunghi, *spatial*, per gestire coordinate).

Possiamo: definire la dimensione di un blocco della chiave con KEY_BLOCK_SIZE, definire qual è il tipo di indice che stiamo usando con USING (se btree, hash o rtree) e inserire un commento con COMMENT.

```
CREATE [OR REPLACE] [UNIQUE|FULLTEXT|SPATIAL] INDEX
[IF NOT EXISTS] index_name
[index_type]
ON tbl_name (index_col_name,...)
[index_option]

index_col_name:
col_name [(length)] [ASC | DESC]

index_option:
KEY_BLOCK_SIZE [=] value
| USING {BTREE | HASH | RTREE}
| COMMENT 'string'
```

Eventi temporizzati

Un **evento temporizzato** è un evento che si verifica dopo un po' di tempo o a cadenza periodica. Un evento che viene inserito nel DBMS è un evento one-shot, ossia che viene utilizzato una volta sola; per definire un evento come *davvero* periodico, utilizziamo la dicitura on completion preserve.

```
set global event_scheduler = on;

create event if not exists `cleanup`
on schedule
every 2 day
on completion preserve
comment 'Remove old tuples'
do
delete from `entity` where `created_at` < (NOW() - interval 2 day)
```

Funzioni

Le **funzioni scalari** sono funzioni a livello di ennupla che restituiscono singoli valori. Abbiamo funzioni:

- temporali: current_date, extract(year from ...);
- manipolazione stringhe: char_length, lower;
- conversione: cast;
- condizionali.

Le **funzioni condizionali** si riassumono in due funzioni principali case (per implementare uno switch-case sui valori di determinati attributi) e coalesce (che permette di definire una serie di attributi: il DBMS dovrà restituirci il primo tra quegli attributi che è non nullo).

```
select Nome, Cognome, coalesce(Dipart,'Ignoto')
from Impiegato

select Targa,
case Tipo
when 'Auto' then 2.58 * KWatt
when 'Moto' then (22.00 + 1.00 * KWatt)
else null
end as Tassa
from Veicolo
where Anno > 1975
```

Possiamo anche definire delle *nostre* funzioni, invocabili negli statement di select. Vediamo un esempio in cui vogliamo validare che l'e-mail inserita sia corretta:


```

delimiter !
create function `validate_email`(email varchar(45))
returns bool
deterministic
begin
    if email regexp '^[a-zA-Z0-9][a-zA-Z0-9._-]*[a-zA-Z0-9._-]@[a-zA-Z0-9][a-zA-Z0-9._-]*[a-zA-Z0-9]\\.[a-zA-Z]{2,63}$' then
        return true;
    end if;
    return false;
end!
delimiter ;

```

La keyword `deterministic` dice che la funzione che stiamo definendo è pura: se non modifico l'input, l'output non verrà modificato.

Con la keyword `delimiter` possiamo comunicare al parser dell'SQL che cambiamo il delimitatore di fine riga (in questo caso diventerà "!").

Creazione Stored Procedure

La **procedura** (*stored procedure*) è un insieme di statement SQL che permette di eseguire qualunque comando SQL al suo interno, ma ci consente di avere come valore di ritorno un result set. Avendo come valore di ritorno un insieme di dati, ci potremmo trovare nella situazione in cui vorremo avere un vero e proprio valore di ritorno (magari per sapere se l'operazione è andata a buon fine oppure no).

```

CREATE
    [OR REPLACE]
    PROCEDURE sp_name ([proc_parameter[,...]])
    [characteristic ...] routine_body

proc_parameter:
    [ IN | OUT | INOUT ] param_name type

characteristic:
    LANGUAGE SQL
    | [NOT] DETERMINISTIC
    | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
    | COMMENT 'string'

routine_body:
    Valid SQL procedure statement

```

Ciascun parametro, oltre ad avere un nome e un tipo, ha un modificatore (IN, OUT, INOUT). Ogni procedura può essere caratterizzata: `LANGUAGE SQL` (per dire che effettivamente stiamo utilizzando degli statement SQL), `[NOT] DETERMINISTIC` (per dire se la procedura è deterministica o meno), `READS SQL DATA` e `MODIFIES SQL DATA` (per dire, rispettivamente, che la procedura legge dati e modifica dati), `COMMENT`.

Le procedure possono soffrire di anomalie dovute alla concorrenza. Per questo motivo, ha senso progettare le procedure in maniera congiunta con le transazioni. Ad esempio:

```

set autocommit = 0;

create procedure `procedure_name`(in input varchar(45),
                                out output varchar(128))
begin
    declare exit handler for sqlexception
    begin
        rollback; -- rollback any changes made in the transaction
        resignal; -- raise again the sql exception to the caller
    end;

    set transaction isolation level repeatable read;
    start transaction;
    if ... then
        signal sqlstate '45001' set message_text = "Si è verificata una condizione di errore";
    end if;
    commit;
end

```


Con `start transaction` stiamo dicendo che da quel punto in poi fino al `commit` tutto quello che si trova all'interno deve essere eseguito in maniera indivisibile/atomica: tutto deve essere eseguito correttamente, altrimenti non bisogna lasciare traccia sul DBMS.

Le transazioni possono essere soggette a deadlock e per questo motivo ogni volta che parte una transazione il DBMS fa partire un cronometro: se la transazione dura più di quanto prestabilito, il DBMS fa terminare la transazione.

Livelli di Isolamento

Quando andiamo a costruire delle procedure transazionali (sia che esse leggano, scrivano o leggano e scrivano) dobbiamo costruirvi intorno delle “bolle” che dicano al DBMS in che modo una transazione deve interagire con possibili altre transazioni concorrenti. Questo concetto si chiama **isolamento delle transazioni**: se abbiamo due flussi di esecuzione transazionali in concorrenza e si verificano delle operazioni concorrenti di lettura e scrittura che operano sullo stesso insieme di dati, l'isolamento di una transazione ci dice fino a che punto essa può “soffrire” di aggiornamenti fatti da altri.

Lo standard SQL implementa quattro livelli di isolamento:

1. **READ UNCOMMITTED** – La procedura che stiamo proteggendo non ha nessun tipo di problema anche se legge dei dati che sono stati scritti da una procedura che non è ancora andata in commit.
L'eseguire un'operazione di `SELECT` non va a verificare se il lock sulla riga è stato preso oppure no → gli statement `SELECT` sono eseguiti in modalità non bloccante, ma è possibile che venga utilizzata una versione precedente di una riga. Pertanto, utilizzando questo livello di isolamento, si può verificare la prima anomalia, nota come **dirty read**: le letture possono non essere coerenti.
2. **READ COMMITTED** – Il DBMS acquisisce un lock per ogni dato che viene letto o scritto. I lock associati ai dati che sono stati aggiornati in scrittura sono mantenuti fino alla fine della transazione, mentre i lock associati ai dati acceduti in lettura sono rilasciati alla fine della singola lettura. Possono verificarsi anomalie di tipo **unrepeatable reads**: non possiamo effettuare letture ripetibili, in quanto se rileggiamo non abbiamo la garanzia di rileggere lo stesso valore.
3. **REPEATABLE READ** – Con questo livello di isolamento, vengono mantenuti i lock sia dei dati acceduti in lettura sia in scrittura fino alla fine della transazione. Non vengono però gestiti i *range lock* (è un lock che si porta dietro un'informazione sull'intervallo dei dati legato a una specifica query di ricerca), pertanto possono verificarsi anomalie di tipo **phantom read**, associate ad inserimenti che avvengono in concorrenza.
4. **SERIALIZABLE** – Tutti i lock vengono mantenuti fino alla fine della transazione e ogni volta che una `SELECT` utilizza uno specificatore di tipo `WHERE`, viene acquisito anche il *range lock*. In sistemi non basati su lock, questo livello di isolamento può essere implementato mediante il concetto di **read/write set**: anziché effettuare il

lock di una tupla, se andiamo a leggere una tupla o un range di tuple in realtà ne facciamo una copia, che diventa una nostra copia privata per la nostra transazione; inoltre, ogni volta che facciamo una scrittura non andiamo a scrivere direttamente sul DBMS, ma su una copia locale.

Quale livello di isolamento scegliere?

- Per molte applicazioni, la maggior parte delle transazioni possono essere costruite in maniera tale da non richiedere l'utilizzo di livelli di isolamento molto alti (ad esempio SERIALIZABLE), riducendo l'overhead dovuto ai lock.
- Lo sviluppatore deve accertarsi con cautela che le modalità di accesso delle transazioni non causino bug software dovuti alla concorrenza e al rilassamento dei livelli di isolamento.
- Se si utilizzano unicamente alti livelli di isolamento, la probabilità di *deadlock* cresce notevolmente.

Modalità di accesso transazionali

READ ONLY: la transazione si limita alla lettura di dati. READ WRITE: la transazione legge e scrive dati. Dare queste informazioni al DBMS consente di generare dei piani di esecuzione ottimizzati dal punto di vista dell'acquisizione dei lock. Sono informazioni ortogonali al livello di isolamento richiesto.

Come marcare le transazioni

```
SET [GLOBAL | SESSION] TRANSACTION
    transaction_characteristic [,transaction_characteristic] ...

transaction_characteristic: {
    ISOLATION LEVEL level
    | access_mode
}
level: {
    REPEATABLE READ
    | READ COMMITTED
    | READ UNCOMMITTED
    | SERIALIZABLE
}
access_mode: {
    READ WRITE
    | READ ONLY
}
```

Nel caso in cui la marcatura non sia globale, le caratteristiche si riferiscono unicamente alla successiva transazione nella sessione. In seguito, verranno ripristinati i valori globali anche per la sessione in corso.

Cursori

```
create procedure `iterate_on_table`()
begin
    declare done int default false;
    declare var_nome varchar(45);
    declare cur cursor for select nome from tabella;
    declare continue handler for not found set done = true;
    open cur;
    read_loop: loop
        fetch cur into var_nome;
        if done then
            leave read_loop;
        end if;
    end loop;
    close cur;
end
```

Il **cursore** è un tipo di SQL che ci permette di scandire riga per riga una tabella associata (nel momento in cui dichiariamo il cursore, la query non è ancora stata eseguita). Possiamo “aprire” il cursore: chiediamo al DBMS di eseguire la query, mantenere in memoria il suo risultato e mantenere un riferimento (ossia il cursore) alla prima tupla risultante da quella tabella.

Per estrarre i dati da una singola riga utilizziamo il comando **fetch**, in cui specifichiamo (con `into`) dove vogliamo che questi dati vengano salvati. C'è da fare attenzione: il numero di attributi che recuperiamo con la `select` deve corrispondere in numero alle variabili che usiamo nella `fetch`.

Possiamo inserire questa operazione di `fetch` all'interno di un `loop`, a cui possiamo anche dare un nome (in questo caso `read_loop`), che chiuderemo con `end loop`.

Quando arriviamo alla fine della tabella, il cursore arrivare a puntare ad una tupla che non esiste: viene sollevata un'eccezione, che gestiamo con un gestore di segnale in cui specifichiamo che la variabile `done` deve assumere valore `true`. Facciamo ciò per sapere quando far finire il ciclo.

Infine, possiamo andare ad utilizzare lo stesso cursore più volte, a patto poi di chiuderlo nel momento in cui non ci è più utile.

L'utilizzo dei cursori porta con sé alcune scelte progettuali che sono significative, come il dover usare un livello di isolamento alto (*serializable*); quindi, dobbiamo utilizzarli se e solo se siamo sicuri che non c'è nessun'altra possibilità, in quanto altrimenti rischiamo di far degradare significativamente le prestazioni del nostro sistema.

Prepared statement

SQL injection è una tecnica di *code injection* usata per attaccare applicazioni che gestiscono dati attraverso database relazionali sfruttando il linguaggio SQL. Infatti, il mancato controllo dell'input dell'utente permette di inserire artificiosamente delle stringhe di codice SQL che saranno eseguite dall'applicazione server (rischiando di causare gravi danni).

Ci sono due modi per risolvere questo problema:

- una connessione deve eseguire un solo statement per volta;
- utilizzare il *Principle of List Privilege*, ovvero un costrutto fondamentale della sicurezza informatica: se stiamo eseguendo un'operazione, dobbiamo avere un insieme di privilegi che sia l'insieme più piccolo per eseguire tutte e sole le operazioni che dobbiamo eseguire.

I **prepared statement** sono una tecnica che consente di inviare al DBMS una *query incompleta*, ossia con dei **placeholder** (rappresentati dal carattere ?), al fine di differenziare le stringhe di comando da quelle di parametro.

```
INSERT INTO score (event_id,student_id,score) VALUES(?,?,?)
```

Il DBMS analizza, verifica e compila tale query. Successivamente, si possono “collegare” a tale query dei parametri; quando il DBMS riceve questi parametri, esegue la query completa e restituisce i risultati. Questa tecnica risolve i problemi di SQL injection e può migliorare le prestazioni delle applicazioni se la stessa query viene eseguita spesso dall'applicazione.

Gestione di prepared statement

PreparedStatement è una classe figlia di `Statement` (già vista in `jdbc`) e deve essere creato su una connessione. I passi per usare i prepared statement sono:

- creazione dello statement a partire da una stringa con placeholder;
- inserimento dei valori;
- esecuzione dello statement;
- processamento del result set.

Vediamo un esempio di gestione di lettura:

```
PreparedStatement stmt = con.prepareStatement(
    "select Stipendio from Impiegato where Cognome = ?");
stmt.setString(1, "Bianchi");
ResultSet rs = stmt.executeQuery();
while (rs.next()) {
    int stipendio = rs.getInt(1);
    System.out.println("Stipendio = " + stipendio);
}
rs.close();
stmt.close();
```

È importante che ci sia un matching (in `setString`) tra il numero di “?”, il tipo che vogliamo inserire (quindi, in questo caso, `String`) e il valore inserito.

Vediamo un esempio di gestione di scrittura, dove la variabile intera *n* rappresenta il numero di righe effettivamente coinvolte nell'operazione di `update`.

```
String updateString =
    "update Impiegato set Stipendio = ? where Cognome = ?";
PreparedStatement updateSalary =
    con.prepareStatement(updateString);
updateSalary.setInt(1, 2500);
updateSalary.setString(2, "Bianchi");
int n = updateSalary.executeUpdate(); // n: Affected rows
updateSalary.close();
```

Invocazione di Stored Procedure

In JDBC, utilizziamo l'interfaccia `CallableStatement`, che estende `PreparedStatement`, per invocare stored procedure.

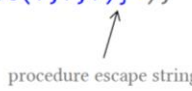
Il passaggio di parametri avviene come per i prepared statement: i parametri in output vanno registrati con **`registerOutParameter()`** e i parametri INOUT devono sia ricevere un valore che essere registrati.

I valori dei parametri in output possono essere recuperati con i metodi `getXXX()`. Le invocazioni devono avvenire *dopo* aver processato le tabelle del result set, al fine di garantire la massima portabilità verso qualsiasi DBMS.

Un esempio è il seguente:

```
CallableStatement cs = con.prepareCall("{call proc(?,?,?)}");
cs.setString(1, "argument");
cs.setFloat(2, 1.);
cs.registerOutParameter(3, Types.NUMERIC);
cs.setFloat(3, 123.45);
ResultSet rs = cs.executeQuery();

while (rs.next()) {
    ...
}
float f = cs.getFloat(3);
```



Notiamo come ogni statement di `CallableStatement` debba essere inserito all'interno di parentesi graffe. Nelle righe 2-3 andiamo a “valorizzare” i placeholder, mentre nelle righe 4-5 valorizziamo il terzo parametro che, essendo INOUT, va trattato in entrambi i modi.

Terminato il ciclo arriviamo alla fine del result set: possiamo ora recuperare un parametro in uscita.