UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II

Facoltà di Ingegneria
Corso di Studi in Ingegneria Informatica

Relazione per il corso di Basi di dati II

Prof. Lucio Sansone

Utilizzo degli indici nei DBMS relazionali

Giuseppe Di Luca - Matr. 885-326
Vincenzo Ciriello - Matr. 885-327

Indice

Capitolo 4. Esempi di utilizzo degli indici

Capitolo 1

Organizzazione delle basi di dati

1.1 Introduzione alle problematiche di organizzazione fisica

In questi paragrafi si espongono le caratteristiche che un buon DBMS dovrebbe avere tenendo in conto le problematiche cui far fronte nell'ambito del recupero dei dati, garantendo affidabilità e sicurezza agli utenti della base dei dati stessa. Si discuterà pertanto di tali problematiche accennando ad alcune strategie impiegate per la loro risoluzione, nonché dell'organizzazione dei file della base di dati sui dispositivi di memoria secondaria e dunque dei metodi di accesso implementati al fine di garantire un rapido recupero delle informazioni.

1.1.1 Caratteristiche della memoria secondaria

Le basi di dati hanno la necessità di gestire le informazioni in esse contenute in memoria secondaria per due motivi fondamentali:

1. dimensione della memoria principale disponibile non sempre sufficiente a contenere tutte le informazioni contenute nella base di dati;
2. non persistenza della memoria principale.

In effetti si può affermare che il motivo predominante nell'utilizzo della memoria secondaria è certamente il secondo fra i due precedentemente esposti. Si ricorda che la memoria secondaria è chiamata in questo modo perché non direttamente utilizzabile dai programmi: i dati, per poter essere utilizzati, devono inizialmente essere trasferiti in memoria principale. Inoltre, si fa presente che i meccanismi di memorizzazione forniti dai moderni sistemi operativi consentono di immagazzinare le informazioni nei dispositivi di memoria secondaria secondo una disposizione in blocchi di dimensione fissata. I sistemi stessi, dunque, definiscono come uniche operazioni possibili la lettura e la scrittura di un blocco. Senza scendere nei dettagli di costo dell'accesso alla singola informazione (bit), appare evidente che il costo di un'operazione di lettura o scrittura nella base dati può essere assimilato praticamente per intero al costo dell'operazione (in termini di tempo) che viene effettuata sulla memoria secondaria: il dispositivo di memorizzazione persistente di una base di dati diviene, pertanto, un "collo di bottiglia" per le prestazioni dell'applicazione di gestione dei dati.

Nel seguito di questo capitolo descriveremo alcune tecniche sviluppate nel corso del tempo al fine di migliorare le performances nella gestione delle basi di dati.

1.1.2 Utilizzo e gestione del buffer

Una prima strategia implementata dai DBMS odierni è la *bufferizzazione*. Tale tecnica prevede che l'interazione fra la memoria principale e quella secondaria sia realizzata nei DBMS attraverso l'utilizzo di una apposita grande zona di memoria centrale detta *buffer*. Tale strategia consente (anche grazie al fatto che negli ultimi anni la memoria centrale ha incrementato notevolmente la sua capacità) di evitare la ripetizione di accessi alla memoria secondaria qualora un dato debba essere utilizzato più volte in tempi ravvicinati.

In questa sede ci limitiamo ad asserire che il buffer è organizzato in *pagine*, che hanno dimensione pari a un numero intero di blocchi del disco (la cui dimensione, ricordiamo, è fissata dal sistema operativo). Un componente del DBMS, detto *gestore del buffer*, si occupa del caricamento e scaricamento delle pagine dalla memoria centrale alla memoria di massa. Il meccanismo esposto implementa, dunque, una prima strategia per il miglioramento delle performances nell'accesso alle basi di dati, infatti:

- in caso di **lettura**, se la pagina è già presente nel buffer, allora non è necessario effettuare la lettura fisica (ovvero in memoria secondaria);
- in caso di **scrittura**, il gestore del buffer può decidere di differire la scrittura in memoria di massa ottemperando comunque alle proprietà di affidabilità del sistema.

Si noti che le politiche di gestione del buffer assomigliano a quelle della gestione della memoria centrale da parte dei sistemi operativi, basate sul principio di *"località dei dati"*. Questo asserisce che i dati referenziati di recente hanno maggior probabilità di essere referenziati nuovamente nel futuro, e si ispirano alla legge empirica dell'*80-20.*[1] Tuttavia, com'è lecito attendersi, la sola politica di bufferizzazione risulta alquanto insufficiente, in quanto non sempre i dati necessari per una specifica elaborazione sono effettivamente subito disponibili in memoria centrale.

1.2 Le strutture di accesso ai file

Attualmente i DBMS utilizzano i file system messi a disposizione dai sistemi operativi, ma creano al contempo stesso una propria astrazione dei files per garantire efficienza e

[1] Essa afferma che il 20% dei dati è tipicamente acceduto dall'80% delle applicazioni. Tale legge ha come conseguenza il fatto che generalmente i buffer contengono le pagine cui viene fatta la maggior parte degli accessi.

transazionalità. In particolare è opportuno osservare che, nella maggior parte dei casi, i DBMS utilizzano solo poche funzionalità di base offerte dal sistema operativo, avvalendosi della possibilità di creare ed eliminare file, o leggere e scrivere singoli blocchi e sequenze di blocchi contigui. Nel seguito vediamo, brevemente, come i DBMS dispongono le tuple delle varie relazioni nei blocchi costituenti la memoria secondaria.

1.2.1 Cenni sulle strutture di accesso primarie

Nel caso più frequente ogni blocco del dispositivo di memoria secondaria è dedicato a tuple di un'unica relazione, ma esistono tecniche che prevedono la memorizzazione delle tuple di più tabelle, fra loro correlate, negli stessi blocchi. In ogni caso, definiamo *struttura di accesso primaria* di un file la struttura che stabilisce il criterio secondo il quale le tuple sono disposte nell'ambito del file. Le strutture di accesso primarie possono essere divise in tre categorie principali:

- ***sequenziali***: possono essere suddivise in tre ulteriori categorie, ovvero:
  - *seriali*: anche dette *entry-sequenced* o *heap*. In esse i dati sono "ammucchiati" nel file senza un ordine particolare, e la sequenza di tuple è indotta dal loro ordine di immissione;
  - *ad array*: le tuple sono disposte come in un array e la loro posizione dipende dal valore assunto in ciascuna tupla da un campo detto *indice*;
  - *ordinate*: la sequenza delle tuple dipende dal valore assunto in ciascuna tupla da un campo di ordinamento detto *campo chiave*;
- ***ad accesso calcolato***: queste garantiscono, al pari della struttura sequenziale ordinata, un accesso associativo ai dati, in cui cioè la locazione fisica dei dati dipende dal valore assunto da un campo chiave. In pratica è possibile pensare di trasformare i valori della chiave in possibili indici di un array attraverso una funzione detta di *hashing*: ciò rende questa struttura primaria estremamente efficiente per accessi di tipo puntuale, ma del tutto inadeguata per accessi su intervalli di valori;
- ***ad albero***: si rimanda al prossimo paragrafo per una trattazione estesa sulle strutture di accesso ad albero. Ci limitiamo per il momento ad asserire che, rispetto alle strutture di accesso precedentemente menzionate, le strutture ad albero risultano essere molto efficienti sia per accessi puntuali che per accessi su intervalli di valori.

1.3 Le strutture ad albero

In questo paragrafo analizziamo le strutture di accesso ad albero chiamate *indici*. Le organizzazioni ad albero possono essere utilizzate per realizzare sia strutture primarie, cioè strutture atte a contenere i dati (come visto al paragrafo 1.2.1), sia strutture secondarie, cioè che favoriscano gli accessi ai dati senza contenere i dati stessi. In quanto tali, le strutture secondarie ad albero possono essere aggiunte ed eliminate dinamicamente ad una base di dati.

Per quanto si possa utilizzare queste organizzazioni per l'accesso primario ai files, si può comunque affermare che l'impiego più comune degli indici consiste nel fornire percorsi di accesso secondari, offrendo metodi alternativi per accedere ai record senza influenzare la loro posizione fisica sui dispositivi di memoria secondaria.

Gli indici consentono un accesso efficace sulla base di campi specifici, detti di *indicizzazione*. Qualsiasi campo del file può essere utilizzato per creare un indice, e sullo stesso file possono essere costruiti più indici su campi differenti ciascuno dei quali, magari, implementato secondo una particolare struttura dati al fine di velocizzare la ricerca.

L'idea su cui si basa la struttura di accesso degli indici ordinati è simile a quella che sta alla base dell'indice analitico di un libro di testo, che riporta in ordine alfabetico concetti fondamentali con l'indicazione del numero di pagina in cui compaiono. La ricerca nell'indice di una base di dati, in perfetta analogia con quella di un libro di testo, permette di consultare un elenco di puntatori a blocchi (o record), consentendo una rapida individuazione dei dati riferiti al particolare valore contenuto nel campo di indicizzazione. Si può ben comprendere che l'alternativa, in mancanza di un indice, sarebbe quella di eseguire una ricerca lineare nel file.

L'indice di solito memorizza ogni valore del campo di indicizzazione corredandolo di un elenco di puntatori a tutti i blocchi del disco che contengono record con quel valore del campo; i valori nell'indice sono ordinati in modo tale da poter eseguire una ricerca binaria.

Facciamo notare, infine, che il file dell'indice è ovviamente molto più piccolo rispetto al file dei dati rendendo dunque una ricerca binaria molto efficiente.

Nei prossimi paragrafi descriviamo le varie tipologie di indici definibili su una base di dati, per poi analizzare brevemente come questi vengano implementati mediante strutture dati ad albero.

1.3.1 Indici primari

Un indice primario è un file ordinato i cui record sono di lunghezza fissa e sono costituiti da due campi. Il primo campo è dello stesso tipo di dati del campo chiave di ordinamento del file di dati, chiamato *chiave primaria*, e il secondo campo è un indirizzo di blocco. Esiste dunque un record nel file dell'indice per ogni blocco del file di dati. Ogni record dell'indice, come si può vedere in fig. 1.1, è composto da due campi:

1. il valore del campo della chiave primaria del primo record del blocco;
2. un puntatore al corrispondente blocco su disco.

Il numero complessivo di tuple dell'indice, pertanto, è uguale al numero di blocchi su disco del file di dati ordinato. Il primo record in ciascun blocco del file di dati è chiamato *record àncora* del blocco o semplicemente *punto àncora*.

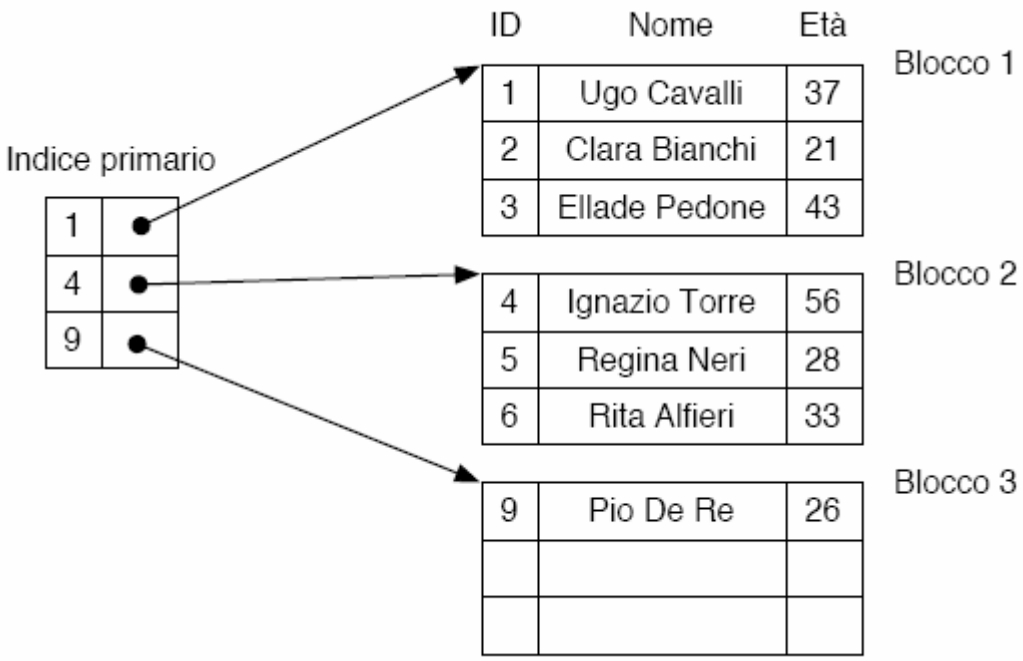


Figura 1.1 – Indice Primario

Gli indici di questo tipo possono essere:

- ***densi***: se contengono un record per ogni valore della chiave di ricerca (e quindi ogni record) del file di dati;
- ***sparsi***: se contengono record solo per alcuni valori di ricerca del file di dati.

In virtù di queste considerazioni, l'indice primario è sparso, poiché include una tupla per ogni blocco del file di dati e le chiavi del suo record àncora, piuttosto che per ogni valore di ricerca o per ogni record.

Il file che memorizza le tuple di un indice primario occupa meno blocchi rispetto al file dei dati per due motivi:

- vi sono meno tuple rispetto al file che contiene i dati effettivi;
- normalmente ogni tupla è di dimensioni inferiori rispetto a un record del file di dati perché ha solo due campi (la chiave di ricerca e il puntatore al blocco o al record corrispondente);

Per questi motivi, un blocco su disco può contenere più record dell'indice che record di dati. Si noti, dunque, che una ricerca binaria nel file dell'indice richiede un numero minore di accessi ai blocchi (e quindi al disco) rispetto ad una ricerca binaria nel file di dati.[2]

Pur risultando estremamente conveniente dal punto di vista della riduzione di accessi alla memoria secondaria, l'indice primario, così come qualsiasi file ordinato, presenta problemi all'atto dell'inserimento e dell'eliminazione dei record nel file di dati. Si osservi che, infatti, se si prova ad inserire un record nella sua posizione corretta (in ordine) nel file di dati, si devono spostare non solo gli altri record per creare spazio a quello nuovo, ma occorre anche cambiare alcuni record dell'indice perché lo spostamento delle tuple nel file di dati potrebbe aver modificato i punti àncora di alcuni blocchi. Questo problema può essere limitato utilizzando un file di overflow non ordinato, oppure utilizzando una lista di record di overflow per ciascun blocco del file di dati. I record all'interno di ogni blocco e la corrispondente lista di overflow possono poi essere ordinati per ridurre ulteriormente il tempo di recupero delle informazioni. Al pari dell'inserimento anche l'eliminazione, causando un riordinamento delle tuple all'interno del file di dati, può comportare dei problemi; tuttavia questi possono essere completamente gestiti utilizzando degli indicatori di eliminazione per ogni singola tupla.

1.3.2 Indici di cluster

Se i record di un file sono ordinati fisicamente rispetto a un campo che non è chiave, cioè che non ha un valore distinto per ciascun record, è possibile definire quel campo *"campo di raggruppamento"*. È possibile creare un apposito tipo di indice, detto *di cluster*, per

[2] Ad esempio, se un file è costituito da b blocchi, una ricerca binaria consta di $log_2 b$ accessi mediamente. Se disponiamo di un indice primario, il cui file è costituito di bi blocchi, con $b_i < b$, la ricerca binaria consterà di soli ($log_2 b_i + 1$) accessi.

velocizzare il recupero dei record che hanno lo stesso valore del campo di raggruppamento. Si noti che in questo caso non è richiesto che il campo di ordinamento abbia un valore distinto per ciascun record.

Anche l'indice di cluster è un file ordinato con due campi:

- uno dello stesso tipo del campo di raggruppamento del file di dati;
- un campo che contiene un indirizzo di blocco.

L'indice di cluster contiene una tupla per ogni valore distinto del campo di raggruppamento. Tale tupla contiene lo specifico valore del campo di raggruppamento e un puntatore al primo blocco nel file di dati che contiene un record con quel valore del campo. Un esempio di indice di cluster è quello mostrato in fig. 1.2.

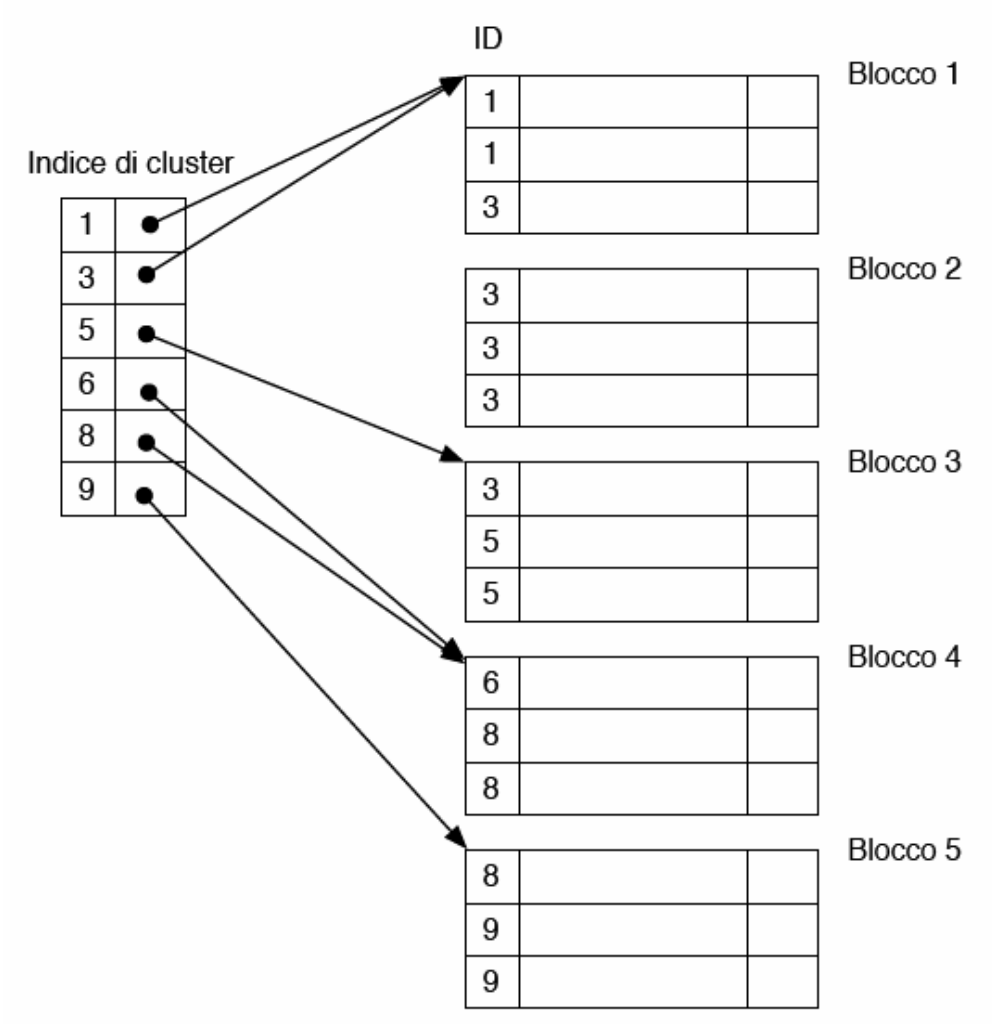


Figura 1.2 – Indice di cluster

Ovviamente, come per l'indice primario, tanto l'inserimento quanto l'eliminazione dei record dal file di dati causano ancora problemi perché questi sono fisicamente ordinati. Una strategia utile a limitare il problema dell'inserimento consiste nel riservare un intero blocco

(oppure un raggruppamento di blocchi contigui) per ciascun valore del campo di raggruppamento: in tal modo tutti i record con quel valore del campo sono posti nel blocco (o raggruppamento di blocchi).

Come si può facilmente osservare, un indice di cluster è sparso perché contiene una tupla per ogni valore distinto del campo di indicizzazione. Dato che tale campo, per definizione, non è una chiave (e quindi ha valori duplicati) non tutte le tuple del file di dati avranno un record corrispondente nel file dell'indice. Facciamo notare, infine, che un file può avere al massimo un campo di ordinamento fisico, quindi si può avere al massimo un indice primario oppure un indice di cluster, ma non entrambi.

1.3.3 Indici secondari

Un indice secondario fornisce un ulteriore strumento di accesso a un file per cui esista già una struttura primaria di accesso. Un indice secondario è essenzialmente un file ordinato costituito da due campi:

- uno dello stesso tipo del campo del file di dati su cui viene definito l'indice (detto *campo di indicizzazione*);
- un secondo campo che contiene un puntatore a blocco oppure un puntatore a record.

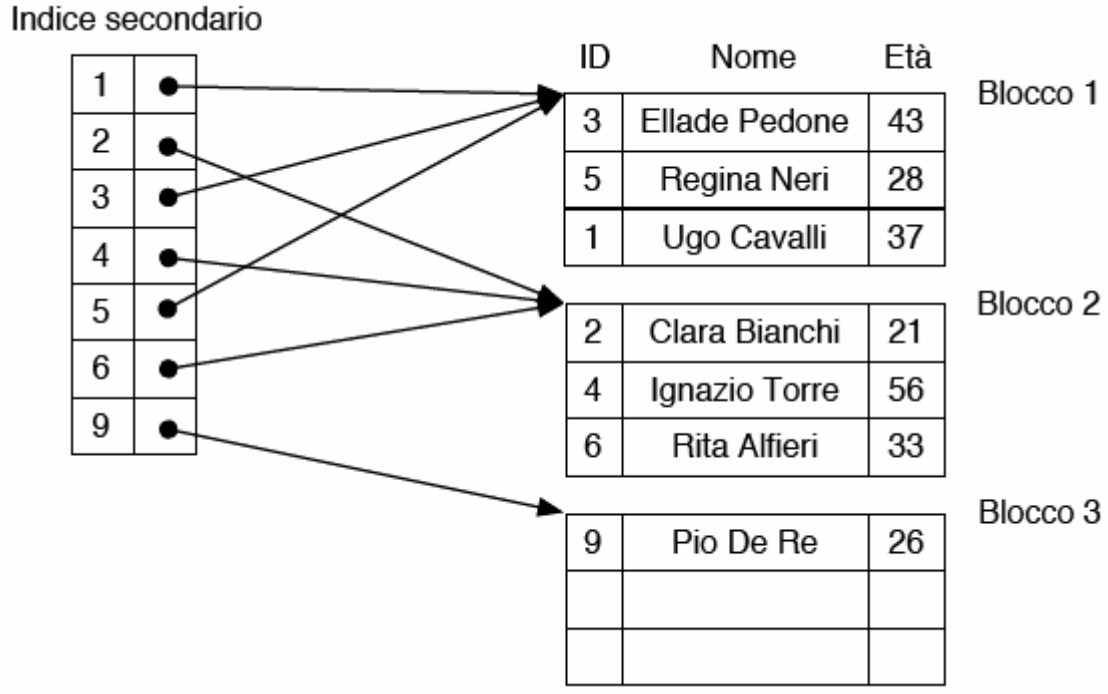


Figura 1.3 – Indice secondario con puntatori ai blocchi

E’ possibile definire molteplici indici secondari per lo stesso file. Si può considerare, ad esempio, la definizione di un indice secondario su un campo (non di ordinamento) che abbia un valore distinto per ciascun record (talvolta un campo del genere viene definito *chiave secondaria*). In questo caso vi è un record dell'indice per ciascuna tupla del file di dati. Tale record è costituito da due campi (come mostrato in fig. 1.3):

- il valore della chiave secondaria della tupla nel file di dati;
- un puntatore al blocco in cui la tupla è memorizzata oppure alla tupla stessa.

Da quanto detto si può facilmente dedurre che un indice secondario è denso. Ovviamente è possibile definire un indice di questo tipo anche su un campo non chiave di una relazione (con valori duplicati). In questo caso numerosi record nel file di dati possono avere lo stesso valore del campo di indicizzazione.

Vi sono varie tecniche per realizzare un indice di questo tipo:

1. inserire più record nell’indice con lo stesso valore del campo chiave (di indicizzazione), uno per ciascun record (del file di dati): essa dà come risultato un indice denso;
2. usare record di lunghezza variabile per i record dell’indice con un campo ricorrente per il puntatore (a blocco). Nel record dell’indice, per il campo chiave (di indicizzazione) viene tenuta una lista di puntatori; la lista contiene un puntatore a ciascun blocco che ospita un record il cui valore del campo di indicizzazione è uguale alla chiave (contenuta nel campo di indicizzazione);
3. mantenere i record dell’indice a una lunghezza fissa e avere un singolo record per ciascun valore del campo di indicizzazione, ma creare un ulteriore livello di gestione e accesso ai puntatori multipli. In questo schema non denso, il puntatore nella generica tupla dell’indice fa riferimento ad un blocco di puntatori ai record (come mostrato in fig. 1.4). Ogni puntatore ai record in quel blocco si riferisce a uno dei record di dati con valore relativo alla chiave per il campo di indicizzazione. Se troppi record condividono lo stesso valore della chiave, così che i loro puntatori ai record non possono essere contenuti in un singolo blocco del disco, viene utilizzato un agglomerato o una lista concatenata di blocchi. Si tratta della tecnica usata più comunemente. Il recupero attraverso l’indice richiede uno o più accessi a blocco aggiuntivi a causa del livello extra, ma gli algoritmi per eseguire ricerche nell’indice e, ancor più importante, per inserire nuovi record nel file di dati sono semplici.

Si osservi che i record dell'indice sono ordinati rispetto al valore del campo chiave, pertanto è possibile eseguire in esso una ricerca binaria; inoltre, stante questo ordinamento, possiamo anche affermare che un indice secondario fornisce un ordinamento logico dei record del file di dati attraverso il campo di indicizzazione.

Facciamo notare che non essendo i record del file di dati ordinati fisicamente rispetto ai valori della chiave secondaria (lo sono invece rispetto ai valori della chiave primaria), non è possibile utilizzare i punti àncora dei blocchi. Per questo motivo viene creato un record dell'indice per ciascun record nel file di dati invece che per ogni blocco come nel caso di un indice primario. In ultima analisi, si osservi che un indice secondario di solito ha bisogno di più spazio di memorizzazione e di un tempo di ricerca più lungo (rispetto a un indice primario) a causa del maggior numero di record contenuti in esso.

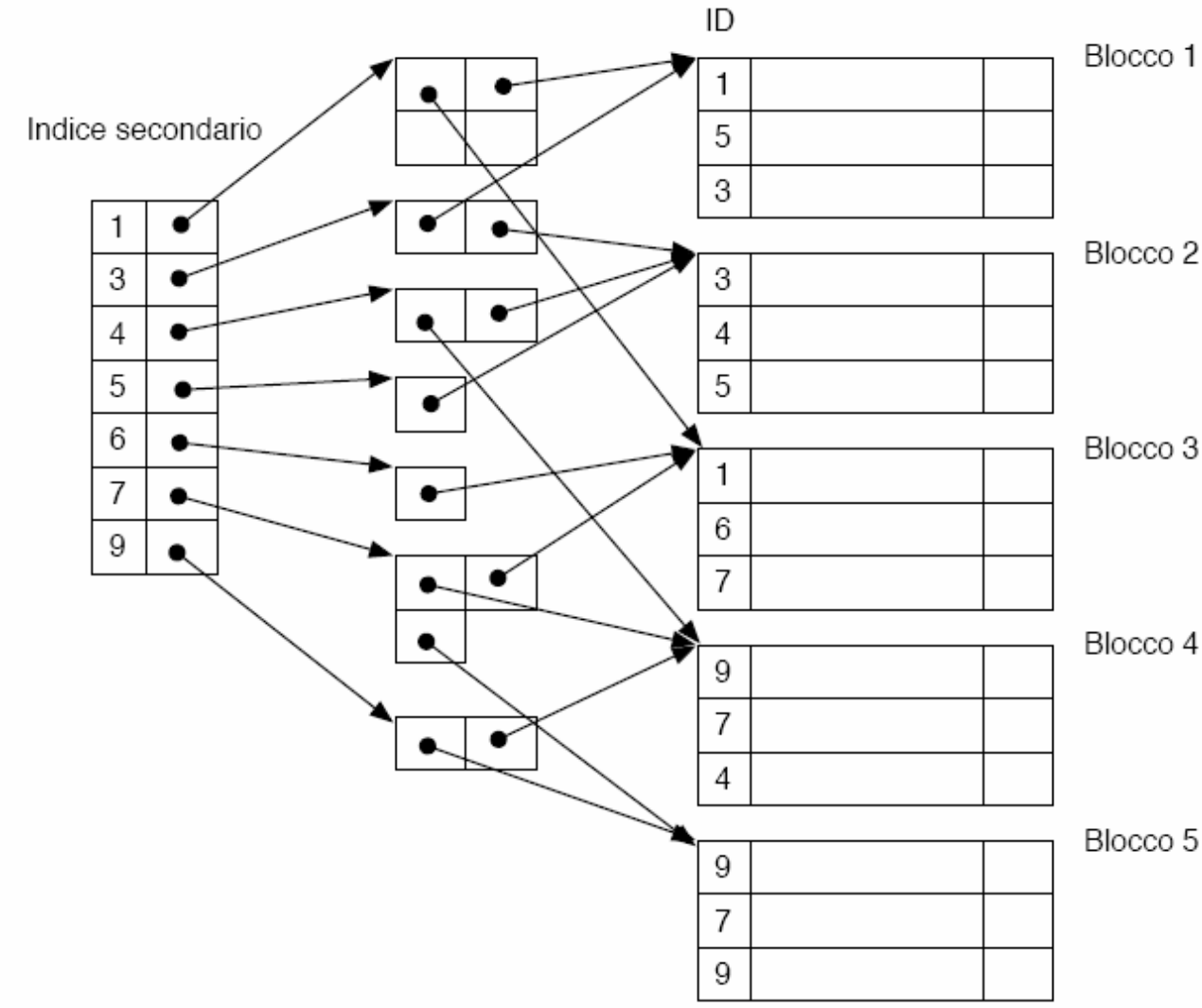


Figura 1.4 – Indice secondario con puntatori a blocchi di puntatori

1.3.4 Indici Multilivello

Gli schemi di indicizzazione presentati nei paragrafi precedenti prevedono che il file dell'indice sia ordinato. Il recupero di un'informazione dal database viene effettuato, pertanto, mediante l'applicazione dell'algoritmo di ricerca binaria sul file dell'indice per recuperare i puntatori ai blocchi o ai record richiesti su disco. Tale algoritmo richiede approssimativamente $log_2 b_i$ accessi al disco per un indice costituito da b_i blocchi.[3]

L'idea che sta alla base di un indice multilivello è quella di ridurre a ogni passo di un fattore F_i la parte dell'indice in cui si continua a cercare, dove F_i rappresenta il fattore di blocco dell'indice (ovviamente maggiore di 2). Il fattore F_i si definisce anche *fan-out* dell'indice multilivello e lo indichiamo nel seguito con il simbolo *"fo"*.

Un indice multilivello considera il file dell'indice, al quale si fa riferimento ora come *indice di primo livello* (o *livello base*), come un file ordinato (di dati) con un valore distinto per ogni chiave. A questo punto è possibile creare un ulteriore indice primario per l'indice di primo livello; questo indice dell'indice di primo livello è detto *secondo livello* dell'indice multilivello. Poiché il secondo livello è un indice primario, si possono utilizzare i punti àncora dei blocchi in modo che il secondo livello contenga un record per ciascun blocco del primo livello. Si noti che il fattore di blocco F_i per il secondo livello e per tutti i livelli successivi è lo stesso che per l'indice di primo livello, perché tutti i record dell'indice hanno la medesima dimensione: ogni tupla dell'indice è infatti costituita da un valore del campo chiave e un indirizzo di blocco.

Consideriamo il seguente caso pratico: se un primo livello di indice è costituito da r_1 record, e il fattore di blocco (che è anche il fan-out) è F_i = fo, il primo livello necessita di $\lceil r_1 / fo \rceil$ blocchi per la memorizzazione su disco. Tale valore risulta essere pari al numero r_2 di record necessari per il secondo livello dell'indice. Possiamo poi ripetere lo stesso ragionamento per il secondo livello. Il terzo livello che è un indice primario per l'indice di secondo livello, ha una tupla per ogni blocco del secondo livello; il numero dei record del terzo livello è quindi $r_3 = \lceil r_2 / fo \rceil$.

Si noti che è richiesto un secondo livello solo se il primo necessita di più di un blocco; analogamente è richiesto un terzo livello solo se il secondo occupa più di un blocco. Si può ripetere il procedimento precedente finché tutte le tuple di un livello *t* dell'indice possono essere contenute in un singolo blocco. Questo blocco di livello *t-esimo* viene chiamato

[3] Quanto scritto trova motivazione nel fatto che ogni passo dell'algoritmo riduce la parte dell'indice in cui si deve continuare a cercare di un fattore pari a 2.

livello superiore (o *livello massimo*) dell'indice. Ogni livello riduce il numero di voci del livello precedente di un fattore fo; pertanto è possibile utilizzare la formula $1 \leq (r_1/(fo)^t)$ per calcolare il numero di livello t di cui sarà costituito l'indice multilivello. In definitiva, un indice multilivello con r_1 record nel primo livello avrà approssimativamente *t* livelli dove $t = \lceil \log_{fo}(r_1) \rceil$. Adottando questo schema, la ricerca in un indice multilivello richiede approssimativamente ($log_{fo} b_i$) accessi ai blocchi, che è un numero inferiore rispetto a quello della semplice ricerca binaria se il fan-out è maggiore di 2.[4] Ciò comporta, dunque, una sensibile riduzione nel numero di accessi ai blocchi (e dunque al disco) durante la ricerca di un record per un dato valore del campo di indicizzazione.

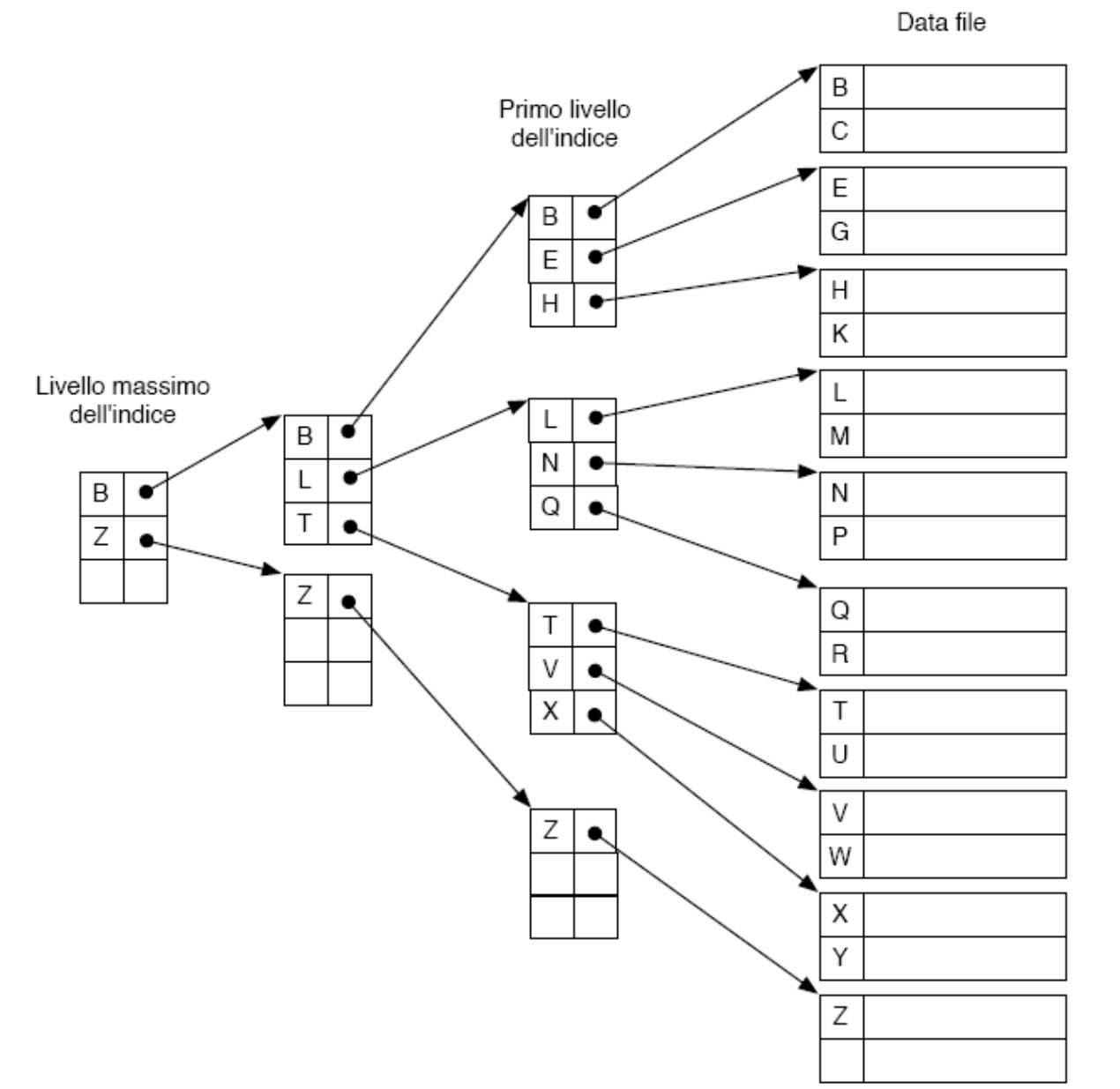


Figura 1.5 – Indice multilivello

[4] Si osservi che ciò è praticamente sempre verificato.

La struttura ad accesso multilivello può essere utilizzata su uno qualsiasi fra i tipi di indici presentati nei paragrafi precedenti, purché l'indice di primo livello abbia due caratteristiche essenziali:

- valori distinti per i campi chiave delle singole tuple;
- record di lunghezza fissa.

In maniera simile a quanto accadeva per altre tipologie di indici, anche per gli indici multilivello occorre affrontare i problemi di gestione delle eliminazioni e degli inserimenti nell'indice, perchè tutti i livelli sono memorizzati in file fisicamente ordinati.

Per mantenere i vantaggi derivanti dall'utilizzo dell'indicizzazione multilivello senza incorrere nei problemi di eliminazione e di inserimento, i progettisti hanno adottato un tipo di indice multilivello che lascia spazio libero in ciascuno dei suoi blocchi per l'immissione di nuove voci. Questo è detto *indice dinamico multilivello* ed è spesso implementato attraverso strutture dati ad albero B o B^+ .

1.3.4.1 L'organizzazione ISAM

Un esempio pratico ed elegante di utilizzo delle strutture ad indice multilivello è rappresentato da un'organizzazione dei dati utilizzata, in particolare, nell'elaborazione gestionale. Tale organizzazione, che consiste in un file ordinato con un indice primario multilivello sul campo di ordinamento, è definita file sequenziale indicizzato (*ISAM – Indexed Sequential Access Method*) ed è stata utilizzata in molti dei primi sistemi IBM. In questa organizzazione, l'inserimento di nuove tuple è gestito attraverso una forma di file di overflow che periodicamente viene fuso nel file di dati ricreando contestualmente l'indice.

L'organizzazione ISAM di IBM, in particolare, sfrutta un indice a due livelli che rispecchia molto da vicino l'organizzazione fisica di un dispositivo di memoria secondaria a disco. Si può osservare, infatti, che il primo livello è un indice di cilindro che contiene per ciascun valore della chiave un record di punti àncora ai cilindri di un'unità disco e un puntatore all'indice delle tracce di ciascun cilindro. L'indice delle tracce, a sua volta, contiene il valore chiave di un record di punti àncora a ciascuna traccia del cilindro e un puntatore per traccia. In ultimo, risulta possibile cercare sequenzialmente il record o il blocco desiderato all'interno della traccia selezionata attraverso l'indice multilivello.

1.3.5 Implementazione degli indici con strutture ad albero (B, B^+, B^*)

Le strutture ad albero dinamiche sono le più frequentemente utilizzate nei DBMS di tipo relazionale per la realizzazione di indici. In questo paragrafo ci limitiamo ad accennare brevemente alle varie strutture possibili, senza entrare nel merito dei dettagli implementativi per ciascuna di esse. Si osservi che ogni albero è caratterizzato da un nodo radice, vari nodi intermedi e nodi foglia; ogni nodo coincide con un blocco (pagina) a livello di file system (gestione del buffer). Gli indici multilivello possono essere pensati come una variante di un albero di ricerca: ogni nodo dell'indice multilivello può avere fino a *fo* puntatori e *fo* valori chiave, dove *fo* è il *fan-out* dell'indice (ovvero il suo *fattore di blocco*). In generale, i valori dei campi dell'indice di ciascun nodo guidano l'utente al nodo successivo finché raggiunge il blocco del file di dati che contiene i record richiesti. Seguendo un puntatore si limita a ogni passo la ricerca a un sottoalbero dell'albero di ricerca e si ignorano tutti i nodi che non sono di quel sottoalbero. I legami tra i vari nodi di un albero vengono stabiliti da puntatori che collegano tra loro le pagine. Un importante requisito per il buon funzionamento delle strutture dati ad albero è che queste siano *bilanciate*, cioè che la lunghezza di un cammino che collega il nodo radice ad un qualunque nodo foglia sia costante. Tale caratteristica garantisce che il tempo di accesso alle informazioni contenute nell'albero sia lo stesso per tutte le foglie, pari quindi alla profondità dell'albero stesso.

Esistono essenzialmente due varianti di strutture ad albero: alberi B e B^+. Un albero B possiede un solo nodo radice (che è anche un nodo foglia) a livello 0. Quando il nodo radice diventa completo e si tenta di inserire un'altra voce nell'albero, il nodo radice si divide in due nodi di livello 1. Analogamente, quando un nodo non radice è completo, e un nuovo record dovrebbe esservi inserito, esso viene diviso in due nodi dello stesso livello e il record centrale ivi contenuto viene spostato verso il nodo padre insieme con i due puntatori ai nuovi nodi ottenuti dalla divisione. La cancellazione di un valore del campo chiave (dai dati effettivi) causa, nell'aggiornamento dell'indice, il fatto che se due nodi adiacenti sono completi per meno della metà (della loro capienza) essi vengano fusi: la cancellazione quindi può ridurre di fatto il numero dei livelli dell'albero. Si noti che gli alberi B sono utilizzati talvolta come struttura primaria per i files di dati (come detto in precedenza). In questo caso tutti i record sono memorizzati all'interno dei nodi dell'albero B invece che solo i record relativi alle singole chiavi di ricerca. Questo funziona bene per i file con un numero relativamente piccolo di record e una dimensione ridotta per ciascuna tupla, altrimenti il fan-out e il numero di livelli diventano troppo grandi per permettere un accesso efficace (ed efficiente). Ciò consente di stabilire che l'utilizzo degli alberi B in qualità di struttura

primaria risulta sconsigliabile in presenza di una grande mole di dati da memorizzare. In definitiva, è possibile affermare che gli alberi B forniscono una struttura di accesso multilivello che produce un albero bilanciato in cui ogni nodo è riempito almeno per la metà.
La distinzione tra alberi B e B^+ è molto semplice: negli alberi B^+ i nodi foglia sono collegati come in una lista che li connette in base all'ordine imposto dal campo di indicizzazione. Tale lista consente di svolgere in modo efficiente anche interrogazioni il cui predicato di selezione definisce un intervallo di valori. Nelle strutture B non viene previsto di collegare sequenzialmente i nodi foglia. In tal caso, un'ottimizzazione possibile prevede di usare nei nodi intermedi due puntatori per ogni valore di chiave K_i; uno dei due puntatori viene utilizzato per puntare direttamente al blocco che contiene la tupla corrispondente a K_i, interrompendo la ricerca; l'altro puntatore serve per proseguire la ricerca nel sottoalbero che comprende i valori di chiave strettamente compresi tra K_i e K_{i+1}. Questa tecnica fa risparmiare spazio nelle pagine dell'indice in quanto ciascun valore di chiave è presente al più una volta nell'albero. L'efficienza di un albero B o B^+ è normalmente elevata, in quanto spesso le pagine che memorizzano i primi livelli dell'albero risiedono nel buffer perchè utilizzate molto spesso.
Si noti che in un albero B tutti i valori del campo di ricerca appaiono una volta a un dato livello nell'albero, insieme con un puntatore ai dati. In un albero B^+ i puntatori ai dati sono memorizzati solo nei nodi foglia dell'albero. Quindi la struttura dei nodi foglia differisce dalla struttura dei nodi interni. I nodi foglia contengono una tupla per ogni valore del campo di ricerca, insieme con un puntatore al record di dati (oppure al blocco che contiene questo record) se il campo di ricerca è un campo chiave. Per un campo di ricerca non chiave il puntatore fa riferimento a un blocco che contiene i puntatori ai record del file di dati creando un livello ulteriore di indici. Come si diceva pocanzi, i nodi foglia sono di solito collegati tra loro per fornire un accesso ordinato al campo di ricerca e ai record: essi sono simili al primo livello (base) di un indice. I nodi interni dell'albero B^+, invece, corrispondono agli altri livelli di un indice multilivello. Alcuni valori del campo di ricerca dai nodi foglia sono ripetuti nei nodi interni dell'albero B^+ per guidare la ricerca. Poiché i record nei nodi interni di un albero B^+ includono valori di ricerca e puntatori a un albero senza alcun puntatore ai dati, un nodo interno di un albero B^+ può contenere più tuple del corrispondente nodo di un albero B. Questo fa sì che l'albero B^+ abbia meno livelli rispetto a un albero B migliorando il tempo di ricerca.
In ultimo, si definisce di solito albero B^* un albero B^+ in cui ogni nodo (che, ricordiamo, rappresenta un blocco) sia completo almeno per i due terzi. È importante osservare questa

definizione, poiché alcuni studi empirici hanno mostrato che in corrispondenza di tale *fattore di riempimento* dei blocchi si ottengono mediamente le migliori prestazioni possibili con le sopracitate strutture ad albero.

1.4 Indici per il Datawarehousing

In questo paragrafo descriviamo due particolari tipologie di indici utilizzate come strutture secondarie per l'accesso ai sistemi di gestione ed analisi dei dati, comunemente chiamati *Datawarehouses*. Non è utile ai fini della nostra discussione una trattazione approfondita di queste particolari basi di dati. Tuttavia è importante comprendere che i datawarehouses si affiancano alle basi di dati atte a contenere i dati effettivi che gestiscono le interrogazioni *OLTP* (*On-Line Transaction Processing*), allo scopo di offrire architetture per l'analisi dei dati a supporto delle decisioni (*DSS - Decision Support Systems*) consentendo la possibilità di effettuare quelle azioni che vanno sotto il nome di *OLAP* (*On-Line Analytical Processing*). Si fa notare, inoltre, che i datawarehouses sono tendenzialmente basi di dati integrate, ovvero i dati in esse contenuti possono provenire da molteplici sorgenti informative; tali dati sono di tipo "storico" e "aggregato" al fine di effettuare stime e valutazioni fuori linea dei dati effettivi. Appare chiaro, dunque, che tali database possono arrivare a contenere una grande quantità di dati (usualmente a sola lettura) e sono fisicamente distinte dalle basi di dati operazionali.[5] Per questi motivi, e anche a causa della complessità delle elaborazioni direzionali su questo tipo di dati, è necessario massimizzare la velocità di esecuzione delle interrogazioni OLAP: l'impiego degli indici in questo contesto assume un'importanza primaria. Di seguito si descrivono le due principali tipologie di indici implementabili allo scopo.

1.4.1 Indici bitmap

Gli *indici bitmap* vengono posti su un attributo di una relazione (detto usualmente campo di indicizzazione) e sono composti da una matrice binaria (i cui possibili elementi sono solamente 0 o 1) che ha le seguenti caratteristiche:

- tante colonne quanti sono i possibili valori dell'attributo;

[5] Per una trattazione più approfondita riguardo i datawarehouse si rimanda ai testi [1] e [2]

- tante righe quante sono le tuple della relazione su cui sono definiti. Per ogni possibile valore dell'attributo, un bit 0/1 indica se il corrispondente record della relazione possiede lo specifico valore.

Generalmente gli indici bitmap sono associati ad alberi B, e consentono una realizzazione efficiente di congiunzioni o disgiunzioni di predicati di selezione, oppure operazioni insiemistiche di unione e intersezione. Mentre la radice e i nodi intermedi di un indice bitmap sono simili a quelli per gli indici tradizionali (per esempio alberi B o B^+), le foglie contengono per ciascun valore dell'indice un vettore i cui bit sono posti a uno in corrispondenza delle tuple che contengono quel valore e a zero altrimenti.

Si può facilmente osservare, dunque, che gli indici bitmap sono particolarmente vantaggiosi nel caso di attributi con pochi valori differenti, e risultano efficienti sia per le query che non richiedono necessariamente un accesso al file di dati che per quelle che utilizzano attributi non molto selettivi nella clausola di selezione (*where*).

Di contro, gli indici bitmap presentano forti limiti quando per un attributo siano possibili un elevato numero di valori differenti, infatti è necessario utilizzare una diversa colonna per ogni possibile valore. Inoltre essi sono difficilmente gestibili se la relazione su cui sono definiti subisce modifiche frequenti, pur rimanendo estremamente convenienti in ambienti come quelli OLAP, nei quali gli aggiornamenti sono piuttosto rari. Si fa notare, ancora, che quando il numero di possibili valori per un attributo è molto alto, la matrice che ne deriva è estremamente sparsa.

Come detto, in definitiva, gli indici bitmap vanno utilizzati solo se i dati sono aggiornati poco frequentemente, in quanto in caso contrario accrescerebbero inutilmente il costo di manipolazione dei dati sulle tabelle che indicizzano.

Se si decide di utilizzare indici bitmap, è necessario considerare i vantaggi nelle prestazioni durante l'esecuzione delle query rispetto agli svantaggi durante i comandi di manipolazione dei dati. Più indici bitmap sono presenti su una tabella, maggiore è il costo nel corso di ogni transazione.

1.4.2 Indici di join

Gli *indici di join* rappresentano un'ulteriore tipologia di indici utilizzata nei sistemi OLAP, talvolta in combinazione con gli indici bitmap presentati al paragrafo precedente. Per descrivere brevemente come vengano implementati questi indici, occorre osservare che

l’analisi dei dati, per l’inserimento in un datawarehouse, avviene rappresentando i dati stessi in forma multidimensionale individuando le seguenti caratteristiche di rilievo:

- ***fatti***: sono i concetti sui quali concentrare l’analisi;
- ***misure***: rappresentano le proprietà atomiche di un fatto;
- ***dimensioni***: descrivono le prospettive lungo le quali effettuare l’analisi dei dati.

È importante osservare queste definizioni perchè la maggior parte dei sistemi OLAP relazionali attualmente in commercio (anche detti *ROLAP – Relational OLAP*) utilizzano una struttura definita *“schema dimensionale”* per la memorizzazione dei dati aggregati all’interno dei datawarehouses. Tale schema prevede che ogni “fatto” sia rappresentato in una relazione, detta comunemente *“tabella dei fatti”*, che memorizza i fatti e le loro misure; associate a tale relazione, lo schema dimensionale propone le varie relazioni rappresentative delle dimensioni di analisi dette *“tabelle dimensione”*. Il compito degli indici di join è appunto quello di consentire una realizzazione efficiente delle operazioni di join tra le tabelle dimensione e la tabella dei fatti. Gli indici di join vengono costruiti sulle chiavi delle tabelle dimensione, e contengono nelle foglie (per ogni valore dell’indice) i puntatori agli insiemi di tuple delle tabelle dei fatti che contengono determinati valori di chiave.
Un particolare tipo di indice di join è lo *“star join index”*: esso estende il join index a più tabelle dimensionali, concatenando colonne relative a più dimensioni al fine di ottenere una tabella di join più complessa.

In definitiva, per le ultime due tipologie di indici presentate (bitmap e join) i costi sono dovuti alla necessità di costruire e memorizzare persistentemente gli indici. I benefici sono legati, invece, al loro uso effettivo da parte del server del datawarehouse per la risoluzione delle interrogazioni.

Capitolo 2

Gli indici in Oracle 10g

2.1 Gestione degli indici

Gli indici, in Oracle 10g, sono strutture opzionali associate alle tabelle. Come qualsiasi altro DBMS di concezione moderna, Oracle rende possibile la creazione di indici per migliorare le performances delle interrogazioni al database; tali strutture forniscono un percorso di accesso rapido alle relazioni contenenti i dati effettivi. Prima di aggiungere indici in Oracle è opportuno esaminare le performances del proprio database, confrontando poi i valori ottenuti con quelli ricavati in presenza di indici. Il DBMS consente la creazione di indici su uno o più campi di una relazione. Dopo la sua creazione, un indice viene automaticamente mantenuto dal DBMS stesso. Gli aggiornamenti della struttura di una tabella o le operazioni sui dati, come l'inserimento, la modifica o l'eliminazione di una tupla, sono automaticamente effettuate per tutti gli indici interessati.

2.1.1 Tipologie di indici

Per mettere a punto gli accessi alle relazioni del database, la versione analizzata di Oracle supporta molti tipi di indici, tra cui i seguenti:

- ***Normal Index***: è l'indice standard, ed è implementato mediante una struttura ad albero B^*. Esso contiene un record per ogni valore del campo di indicizzazione della relazione insieme a un puntatore al record fisico su disco in cui i dati sono effettivamente memorizzati;
- ***Text Index***: è un tipo di indice costituito da un insieme di tabelle e indici mantenuti da Oracle per supportare le esigenze delle ricerche testuali complesse, come ad esempio la ricerca di un testo completo (o parole, frasi) all'interno di pagine Web o documenti;
- ***Single-Column*** e ***Concatenated Indexes***: sono, rispettivamente, gli indici su uno (*single-column index)* o più campi di una relazione (*concatenated index*). I concatenated indexes sono utili quando tutti i campi sui quali è definito l'indice sono utilizzati nella clausola *"where"* di statements SQL eseguiti frequentemente. Per i concatenated indexes è possibile e, anzi, opportuno definire con molta attenzione le priorità di utilizzo (nell'indice) dei campi di indicizzazione, elencandoli in maniera

crescente rispetto al numero di valori duplicati che è possibile trovare in una relazione nello specifico campo di indicizzazione. I campi che presentano molti valori duplicati nell'ambito di una relazione, o contengono in molte tuple il valore *"null"* non dovrebbero essere inclusi, o dovrebbero essere menzionati per ultimi nella definizione dell'indice;

- ***Ascending*** e ***Descending Indexes***: la ricerca di default all'interno di un indice procede in maniera ascendente, ovvero dal valore più piccolo al valore più grande. I caratteri sono ordinati in base al loro valore ASCII, i dati numerici sono ordinati dal numero più piccolo al più grande, e le date sono ordinate dalla più vecchia alla più recente. Questo comportamento, che è reso disponibile di default, viene praticato dagli indici creati come *ascending indexes*. E' possibile invertire l'ordine di ricerca utilizzando, invece, un *descending index*. Questi ultimi sono anche detti *"indici a chiave inversa"* e sono particolarmente convenienti nel caso in cui ci siano conflitti di I/O durante gli inserimenti di valori sequenziali: in tal caso Oracle può invertire dinamicamente i valori indicizzati prima di memorizzarli;
- ***Column*** e ***Function-Based Indexes***: tipicamente, un record dell'indice è basato sui valori del campo di indicizzazione nelle varie tuple di una relazione. Tali indici si definiscono *column indexes*. Alternativamente è possibile creare un *function-based index* in cui il valore indicizzato è derivato dai dati della tabella. Ad esempio, per trovare dati di tipo carattere che possono essere in un case qualsiasi (lower o upper case) è possibile utilizzare un function-based index per ricercare i valori (su cui è definito il campo di indicizzazione) come se essi fossero tutti nello stesso case;
- ***Bitmap Indexes***: per le colonne che contengono alcuni valori unici, un indice bitmap può migliorare le prestazioni delle query. A causa dei meccanismi interni impiegati da Oracle per la loro manutenzione, gli indici bitmap devono essere utilizzati solo quando il caricamento dei dati avviene in modalità batch (come per le applicazioni di datawarehousing);
- ***Partitioned Indexes***: è possibile partizionare gli indici per supportare le tabelle partizionate o per agevolare la gestione degli indici. Le partizioni degli indici possono essere locali per le partizioni delle tabelle o possono applicarsi globalmente a tutte le righe della tabella.

Oracle consente di specificare dove deve essere posto un indice per una tabella assegnando un particolare *tablespace*, che costituisce una particolare sezione logica del database

corrispondente a uno o più file di dati. Logicamente le migliori performances in quanto a gestione e disponibilità si possono ottenere collocando un indice per una tabella in un tablespace situato su un disco fisicamente separato dalla sua tabella corrispondente.

2.1.2 Indici per l'utilizzo con vincoli

Tutti i campi definiti come *"unique"* o *"chiave primaria"* richiedono la realizzazione di un indice corrispondente. Il DBMS di Oracle definisce automaticamente gli indici necessari a supportare l'integrità dei dati su cui siano stati definiti dei vincoli. Ad esempio, se su un attributo è definito un vincolo unique, il DBMS di Oracle crea automaticamente un indice di tipo *"unique key"*.

Oracle definisce, inoltre, le seguenti politiche:

- i vincoli tendono ad utilizzare gli indici esistenti, ove possibile, piuttosto che crearne nuovi;
- su campi di tipo "unique" o "chiave primaria" si possono utilizzare indifferentemente indici "unique" o "non unique";
- al massimo un campo unique o chiave primaria per volta può utilizzare tutti gli indici non unique;
- non c'è bisogno che l'ordine dei campi di indicizzazione nell'indice e i vincoli combacino;
- per scopi di performance, si potrebbero voler aggiungere indici su attributi definiti in tabelle esterne, all'atto della creazione di vincoli di chiave esterna (*foreign key*). Il DBMS di Oracle non effettua questa azione automaticamente, e dunque occorre agire in maniera apposita manualmente.

2.1.3 Linee guida per la creazione di indici

Da quanto visto nel primo capitolo, gli indici garantiscono un accesso più rapido ai dati per le operazioni che restituiscono una piccola porzione dei record di una relazione. In Oracle è possibile creare indici su ogni attributo; comunque, se un attributo non è utilizzato in alcuna interrogazione, appare ovvio che la creazione di un indice su tale attributo risulta deleteria per le performances del database portando via inutilmente delle risorse preziose.

2.1.3.1 Indicizzare le giuste relazioni e i giusti campi

Il reference manual di Oracle, in dipendenza delle caratteristiche specifiche del DBMS implementato nella versione analizzata, fornisce alcune linee guida (che sarebbe opportuno sempre seguire) nella creazione degli indici:

- è auspicabile creare indici sugli attributi di join fra relazioni, al fine di incrementare le prestazioni dell'operazione di join stessa;
- è utile creare indici sui campi di una chiave esterna;
- conviene lasciare senza indici le relazioni "piccole", se non per imporre l'unicità della chiave primaria (una relazione viene considerata piccola quando occupa un numero talmente ridotto di blocchi che Oracle è in grado di leggere tutti i suoi dati con un'unica lettura fisica); è tuttavia possibile comunque definire indici se si osserva un progressivo decadimento delle prestazioni nella velocità di esecuzione di specifiche query (le dimensioni delle relazioni potrebbero essere cresciute).

In particolare sono indicate le seguenti condizioni favorevoli per la definizione di un indice su un campo di una relazione:

- i valori del campo sono **unique**, o ci sono pochi valori duplicati;
- c'è una grande **varietà** di valori;
- il campo contiene, in corrispondenza di tuple differenti, molti valori ***null***, ma le query eseguite selezionano prevalentemente le tuple che invece contengono un valore ***non null***. I campi che contengono molti valori null sono, invece, poco adatti ad essere indicizzati se le ricerche selezionano tuple i cui campi di indicizzazione contengono valori null.

2.1.3.2 Limitare il numero di indici per ogni relazione

È facilmente osservabile che all'aumentare del numero di indici definiti su una relazione, aumenta significativamente il sovraccarico indotto al DBMS quando la relazione stessa viene modificata. Quando le tuple vengono inserite o eliminate, tutti gli indici della relazione devono essere aggiornati; analogo ragionamento si può effettuare quando un qualsiasi campo di una tupla viene modificato. Occorre pesare attentamente i benefici dal punto di vista delle performances ottenuti mediante l'utilizzo degli indici rispetto al sovraccarico indotto dagli aggiornamenti alle relazioni. Per esempio, una buona regola potrebbe essere quella di definire molti indici su una relazione a sola lettura, mentre pochi sulle relazioni aggiornate molto di frequente.

2.1.3.3 Eliminazione degli indici non necessari

È opportuno eliminare un indice se:

- non velocizza l'esecuzione di una query: le prestazioni in presenza e assenza dell'indice risultano dello stesso ordine;
- le interrogazioni alla base di dati non utilizzano il campo di indicizzazione su cui è costruito l'indice: in tal caso l'indice è perfettamente inutile.

Si noti che non è possibile eliminare un indice che è stato creato (in maniera automatica o manuale) su di un vincolo. Occorre dapprima eliminare il vincolo, e in automatico Oracle eliminerà anche l'indice ad esso associato. Se si elimina una relazione, tutti gli indici ad essa associati vengono eliminati di conseguenza.

Capitolo 3

Gli indici in MySQL 6.0

3.1 Scelte di design

MySQL mantiene le tuple dei dati effettivi e quelle degli indici in files separati. La maggior parte (quasi tutti) degli altri sistemi DBMS mescolano i dati effettivi e quelli degli indici nello stesso file. I progettisti di MySQL hanno ritenuto che la scelta effettuata nella progettazione del DBMS risulti adeguata per la gran parte dei sistemi di elaborazione dati moderni.

Un altro modo di immagazzinare i dati utilizzato da MySQL è quello di mantenere le informazioni relative ad ogni colonna in un'area separata. Ciò consente un notevole incremento delle performances relativamente a quelle interrogazioni che accedono a più di una colonna (attributo); tuttavia questo meccanismo degrada le prestazioni del sistema rapidamente e in maniera molto sensibile, e dunque non viene completamente implementato perché ritenuto inadatto per i database di tipo *general-purpose*.

Il caso più comune è quello in cui l'indice e i dati sono memorizzati insieme (come in Oracle/Sybase, et al). In questo caso è possibile trovare le informazioni nella pagina foglia dell'indice. La cosa buona di questo layout è che esso, nella maggior parte dei casi, se la politica di *caching* dell'indice è buona, consente il risparmio di una lettura su disco. I fattori negativi di questo layout sono i seguenti:

- la scansione della tabella è più lenta poiché occorre leggere attraverso l'indice per giungere alle tuple contenenti i dati di interesse;
- non è possibile utilizzare solo la tabella indice per recuperare i dati attraverso un'interrogazione;
- viene utilizzato più spazio su disco poiché bisogna duplicare gli indici tra i nodi (non è possibile memorizzare le tuple nei nodi);
- le cancellazioni degenerano la tabella nel corso del tempo (poiché gli indici nei nodi non sono aggiornati, di solito, all'occorrenza di una cancellazione);
- è più difficile effettuare il caching per i soli dati dell'indice.

3.2 Tipologie di indici

3.2.1 Indici di colonna

Tutti i tipi di dato di MySQL possono essere indicizzati. L'utilizzo degli indici sulle colonne rilevanti è il modo migliore per incrementare le performances sulle operazioni di *"select"*.
Il massimo numero di indici per tabella e la massima lunghezza per indice è definita per ciascun motore di archiviazione (*storage engine*). MySQL supporta diversi storage engines, ovvero delle entità che agiscono come *"handlers"* per differenti tipi di tabelle; dalla versione 5.1 di MySQL è stata introdotta un'architettura di storage engines modulare, che consente il caricamento di nuovi handlers non previsti dalla piattaforma.[6] Tutti gli storage engines supportano almeno 16 indici per tabella e almeno una lunghezza complessiva dell'indice di 256 bytes, ma gran parte degli storage engines ha capacità molto maggiori.
MySQL consente, in particolare, di definire sui campi stringa anche indici che utilizzino solo una porzione di caratteri dell'attributo stesso. L'indicizzazione di una parte ridotta di un attributo di tipo stringa consente una sostanziale riduzione del file dell'indice. Infine è possibile creare indici *"fulltext"* utilizzabili per ricerche di testi interi. È possibile creare indici su dati di tipo spaziale: in particolare esiste lo storage engine MyISAM che supporta indici ad albero R. Gli altri storage engines utilizzano alberi B per indicizzare tipi spaziali. Infine c'è lo storage engine *"memory"* che utilizza indici hash di default, ma supporta anche indici di tipo ad albero B.

3.2.2 Indici multi-attributo

MySQL può creare, inoltre, indici compositi, ovvero costituiti su più colonne (attributi) di una relazione. Un indice può consistere al massimo di 15 colonne. Per alcuni tipi di dato è possibile indicizzare anche solo una parte (prefisso) del dato contenuto nella colonna. Un indice su più colonne può essere considerato come un array ordinato che contiene i valori creati concatenando i valori delle colonne indicizzate. MySQL utilizza indici su più colonne in modo tale che le interrogazioni risultino più veloci qualora si specifichi un valore conosciuto per la prima colonna dell'indice, ad esempio nella clausola *"where"*, anche se non si specificano valori per le altre colonne.

[6] Una trattazione approfondita sugli storage engines esula dagli scopi di questo documento. Per essa si rimanda al Reference Manual di MySql 6.0.

3.3 Modalità di impiego degli indici

Gli indici sono utilizzati per trovare righe che abbiano uno specifico valore in una colonna più velocemente. Senza alcun indice, MySQL deve iniziare dall'inizio con la prima riga e leggere attraverso l'intera tabella per trovare le tuple rilevanti ai fini della ricerca. Più grande è la tabella, maggiore sarà il costo di questa operazione. Se la tabella possiede un indice sul campo specifico MySQL può determinare più velocemente la posizione per vedere all'interno dei dati senza dover leggere tutti i dati. Se una tabella è costituita da 1000 righe, l'operazione è almeno 100 volte più veloce rispetto al caso in cui la si legga sequenzialmente. Se invece si necessita di accedere alla maggior parte delle tuple, è più veloce una lettura sequenziale, poiché così facendo si minimizzano le ricerche su disco. La maggior parte degli indici MySQL (*primary key*, *unique*, *index*, *fulltext*) sono mantenuti attraverso alberi B. Fanno eccezione particolari tipologie di indici, come quelli sui tipi di dati spaziali (mantenuti attraverso alberi R); il motore di archiviazione *"memory"* supporta, infine, anche indici di tipo *"hash"*.
MySQL utilizza gli indici per le seguenti operazioni:

- per trovare velocemente le tuple che soddisfino le condizioni poste nella clausola di *"where"*;
- per non considerare delle tuple; se è possibile scegliere tra più indici su uno specifico campo, MySQL di solito utilizza l'indice che recupera il minor numero di tuple;
- per recuperare tuple da altre relazioni durante operazioni di join. MySQL può utilizzare indici sulle colonne più efficientemente se essi sono dichiarati dello stesso tipo e dimensione.

In alcuni casi, tuttavia, MySQL non utilizza alcun indice, anche se ne è disponibile almeno uno. Una circostanza in cui ciò può accadere è quando l'ottimizzatore stima che l'utilizzo dell'indice porterebbe ad accedere a gran parte delle tuple nella tabella. In questo caso, è semplice osservare che una scansione lineare sarebbe più efficiente, richiedendo un numero minore di accessi al disco.

3.3.1 La strategia del Key-Caching

Ci preme soffermarci solo un attimo su una particolare strategia adottata in MySQL. Per minimizzare l'I/O su disco, lo storage engine MyISAM implementa un'interessante strategia, nota col nome di *key caching*, utile al miglioramento delle prestazioni nelle

interrogazioni. Tale engine impiega un meccanismo di cache per mantenere in memoria i blocchi delle tabelle acceduti più frequentemente:

- per i blocchi dell'indice, viene mantenuta una speciale struttura denominata *key cache* (o *key buffer*). La struttura contiene un numero di buffer in cui sono mantenuti i blocchi dell'indice maggiormente acceduti. È possibile specificare la dimensione del buffer per ciascuna key cache, tenendo conto che la migliore performance per le operazioni di I/O si ottiene quando la dimensione dei buffer è uguale alla dimensione dei buffer di I/O del sistema operativo sottostante;
- per i blocchi contenenti le tuple con i dati delle relazioni, MySQL non utilizza nessuna cache particolare, facendo affidamento sul meccanismo di caching del filesystem implementato dal sistema operativo sottostante.

La strategia di key caching consente di garantire una rapidità d'esecuzione ancora maggiore delle interrogazioni, permettendo inoltre che più threads accedano concorrentemente alla cache: l'accesso condiviso alla key cache permette di incrementare significativamente il throughput del database server. Inoltre è anche possibile istanziare diverse key caches e assegnare le tabelle degli indici a specifiche caches, ciò che consente di risolvere i problemi dovuti alla concorrenza tra più threads nell'accedere alla stessa key cache e quindi i problemi dovuti all'accesso condiviso alle risorse.

Quando la key cache non è operativa, i file dell'indice sono acceduti semplicemente utilizzando il buffering del filesystem offerto dal sistema operativo sottostante (in altre parole i files dell'indice sono acceduti allo stesso modo dei file contenenti i dati effettivi).

Un blocco dell'indice è un'unità di accesso contiguo per i file dell'indice MyISAM. Di solito la dimensione di un blocco dell'indice è uguale alla dimensione dei nodi di un indice di tipo albero B (gli indici sono rappresentati su disco utilizzando un struttura dati di tipo albero B. I nodi in basso nell'albero sono detti "foglie", mentre quelli superiori sono detti "interni"). Tutti i buffer in una key cache sono della stessa dimensione, che può essere uguale, maggiore o minore rispetto a un blocco della tabella indice. In ogni caso, una di queste due dimensioni è multipla dell'altra. Quando un dato deve essere acceduto da un qualsiasi blocco della tabella dell'indice, il database server controlla per prima cosa se è presente in qualcuno dei buffer della key cache. Se il dato è presente, esso viene prelevato dalla key cache piuttosto che dal disco. Analogamente accade per la scrittura, che può essere effettuata in un buffer della key cache piuttosto che direttamente sul disco. La politica di replacement seguita per i buffer della key cache è la strategia *LRU (Least Recently Used)*: nello scegliere un blocco da rimpiazzare, il sistema seleziona il blocco dell'indice utilizzato

meno recentemente. Per rendere più semplice questa scelta, il modulo di key cache mantiene una speciale coda (la *LRU chain*) contenente tutti i blocchi utilizzati. Quando un blocco viene utilizzato, esso è posizionato in fondo alla coda; quando un blocco necessita di essere rimpiazzato, esso viene posto alla testa della coda. In tal modo, quando si necessita di eliminare dei blocchi dal buffer della key cache, quelli posti in testa alla coda sono quelli utilizzati meno recentemente e dunque sono i primi candidati all'eliminazione.

Capitolo 4

Esempi di utilizzo degli indici

4.1 Introduzione ai casi di test

In questo paragrafo saranno presentate alcune semplici applicazioni sull'utilizzo degli indici. In particolare, verrà mostrato come l'utilizzo di un indice permetta, nella maggior parte dei casi, di ridurre il tempo di esecuzione di una interrogazione.

I casi presi in esame sono due: quello di un'interrogazione di selezione puntuale e di una selezione multipla. Il primo rappresenta un esempio in cui l'utilizzo di un indice consente di ottenere una riduzione sul tempo di esecuzione delle query, mentre il secondo mostra come la presenza di un indice, in casi particolari, non apporti alcun miglioramento.

Per la realizzazione dei test è stato utilizzato il DBMS Oracle 10g. Le interrogazioni sono state eseguite su una tabella molto semplice chiamata ANAGRAFICA, costituita da cinque campi: Nome, Cognome, Codice Fiscale (chiave primaria), Età e Sesso. Di seguito viene riportata la dichiarazione SQL della tabella:

```
CREATE TABLE  "ANAGRAFICA"
  (     "NOME" VARCHAR2(20),
        "COGNOME" VARCHAR2(20),
        "CODFISC" VARCHAR2(16),
        "ETÀ" VARCHAR2(2),
        "SESSO" VARCHAR2(1),
        CONSTRAINT "ANAGRAFICA_PK" PRIMARY KEY ("CODFISC") ENABLE
  )
```

Per valutare l'impatto dell'indice sul tempo di esecuzione delle interrogazioni è stata realizzata una applicazione che comunica con il database Oracle tramite i driver *jdbc*. Il diagramma delle classi dell'applicazione realizzata è stato riportato nella figura 4.1. La classe *DBMSConnection* fornisce tutte le funzionalità basilari per la comunicazione con il DBMS, astraendole dai dettagli relativi alla particolare piattaforma utilizzata. I metodi previsti, infatti, permettono di:

- stabilire e chiudere connessioni col database rispettivamente tramite i metodi *connect* e *close_connection*;
- interrogare il database: i metodi *select_query* per l'esecuzione di una query (i tre metodi differiscono solo per il tipo in cui viene restituito il risultato della query), *count_query* per una query di conteggio, *update_query* per modificare il contenuto del

database (*insert*, *update* e *delete*) e infine *Table_info* per conoscere lo schema di una tabella.

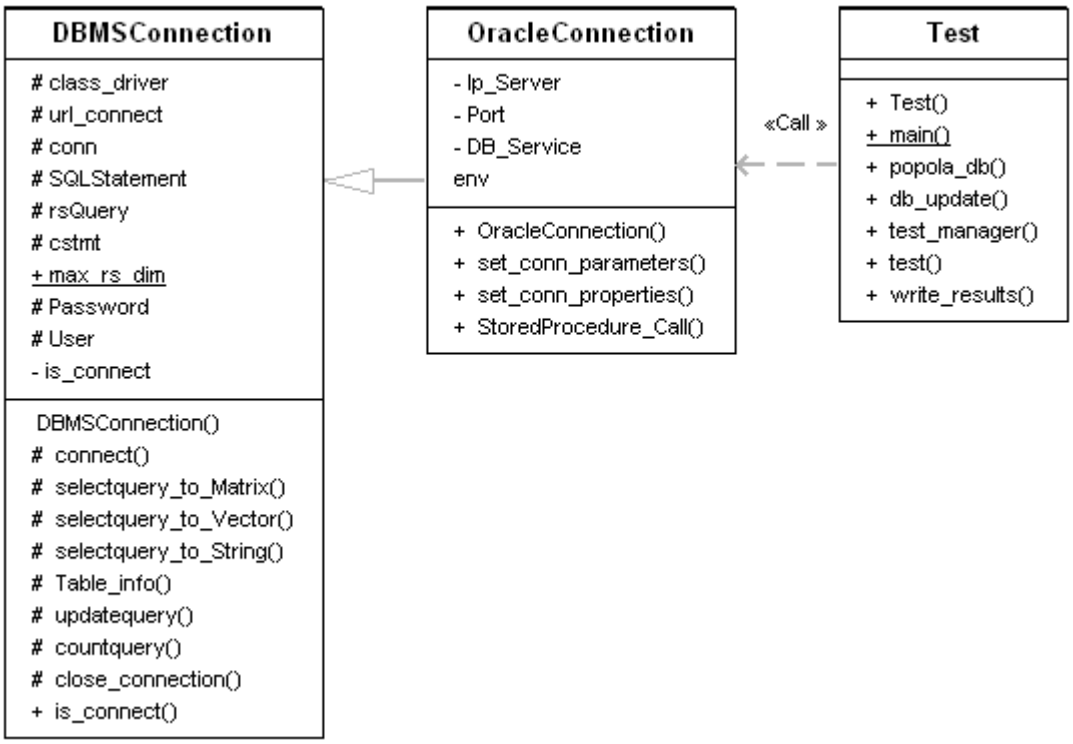


Figura 4.1 Diagramma delle classi dell'applicazione

La classe *OracleConnection* specializza il comportamento della classe *DBMSConnection* per interfacciarsi con il database Oracle. In particolare specifica: i driver utilizzati (*ojdbc1.4* per la versione 10g di Oracle utilizzata), i parametri della connessione (nome del database, indirizzo ip, numero di porto del listener Oracle, e le credenziali di accesso) e infine alcune proprietà, come ad esempio l'autocommit.

Infine la classe Test, richiamando i metodi delle due classi appena descritte, esegue i test progettati che saranno descritti nei prossimi paragrafi. La misura del tempo di esecuzione dell'interrogazione è stata realizzata memorizzando il *timestamp* (con granularità dell'ordine dei nanosecondi) immediatamente prima e dopo l'esecuzione della query sul database. Al fine di ottenere una misura più attendibile, lo stesso test è stato ripetuto diverse volte; nello specifico, ogni valore è stato calcolato come la media su 100 campioni. Inoltre, tutte le tuple della tabella hanno la stessa dimensione.

4.2 Query di selezione puntuale

In questo primo esempio è stato valutato l'impatto dato dall'utilizzo di un indice in una interrogazione di ricerca che seleziona una singola tupla. Nello specifico, la query realizzata prevede una ricerca su campo secondario (Cognome) della tabella. L'obiettivo del test è quello di misurare la variazione nei tempi di esecuzione dell'interrogazione nel caso in cui sia definito o meno un indice sul campo Cognome. Come già detto nel primo capitolo, quando viene condotta una ricerca su di un campo non chiave, il DBMS, in assenza di un indice secondario, effettua la ricerca attraverso una scansione lineare sulla struttura primaria. Per questo motivo la prova è stata realizzata su di una tabella di dimensioni via via crescenti in modo tale da rendere apprezzabile la lentezza della ricerca lineare rispetto a quella in presenza di un indice sul campo di ricerca; infatti, appare evidente che al crescere della tabella una scansione lineare risulti molto più pesante rispetto ad una logaritmica.
Per far in modo che l'interrogazione selezioni una sola tupla, tutti i valori del campo Cognome delle varie tuple sono differenti tra loro. Il meccanismo utilizzato sarà descritto nel prossimo paragrafo.

4.2.1 Descrizione del caso

Le dimensioni della tabella previste per questo test vanno da un minimo di 500 a un massimo di 10000 tuple. Risulta ovvio che per giungere a queste cifre, si è reso necessario realizzare un meccanismo per automatizzare l'inserimento delle tuple all'interno della nostra relazione. Tale compito è stato implementato attraverso il metodo *popola_db* della classe *Test*. Per realizzare valori distinti della chiave primaria è stata utilizzata una stringa alfanumerica composta da una parte alfabetica, comune a tutte le tuple, più una parte numerica variabile. Per rispettare il vincolo che le tuple avessero la stessa lunghezza, è stata utilizzata una parte numerica composta da quattro cifre (che varia da 0000 a 9999), in quanto con quattro cifre si possono ottenere esattamente 10000 le stringhe distinte tra loro. La generazione di questa stringa numerica è stata implementata con un ciclo costituito da quattro *for* innestati, rispettivamente sulle variabili *i*, *j*, *k* e *l*. La stringa costituita dal valore di queste variabili costituirà la parte numerica del campo. Ogni ciclo inizializza la variabile corrispondente a zero e termina quando questa arriva a nove. In questo modo si ottengono tutte le combinazioni da 0000 a 9999. Ad esempio all'inizio, i cicli più esterni su i,k,j e l fissano le corrispondenti variabili a 0, che formano così la stringa 0000; poi il ciclo più interno comincia a far variare l, ottenendo 0001,….0009. Una

volta che il ciclo su *l* si esaurisce, il for immediatamente più esterno incrementa la variabile *k* di 1, e il ciclo su *l* riparte; stavolta però k è pari ad 1 e pertanto si ottengono le stringhe 0010,0011,….,0019; e così via per tutte le altre sino a 9999. Le dimensioni inferiori 7000,5000,2000,1000 sono state ottenute fermando il ciclo su *l* rispettivamente a 7, 5, 2, 1; mentre per 500 sono state utilizzate solo 3 variabili, avendo fissato la prima cifra della stringa, rappresentata dalla variabile *i*, a 0. Lo stesso meccanismo è stato utilizzato per il campo Cognome, ma con una parte alfabetica differente.

La tupla viene inserita nel database tramite il metodo *update_query* di *DBMSConnection*, che prende in ingresso la stringa che rappresenta il comando SQL da eseguire; la chiamata a questo metodo viene eseguita all'interno dei cicli for appena descritti. L'argomento passato è sempre lo stesso e varia solo per i valori delle variabili *i, j, k* e *l*:

insert into ANAGRAFICA values ('Giuseppe','Garibaldi"+i+j+k+l"','FPPGAR2345F"+i+j+k+l"','80','M')

In questo modo ad ogni iterazione del ciclo su *l*, viene inserita una nuova tupla con un valore della stringa composta dalla sequenza delle cifre dei contatori *ijkl*, che costituisce parte del valore del campo Codfisc, di volta in volta distinto.

Dopo aver popolato la tabella, inizia il test vero e proprio richiamando il metodo *test_manager*. Questo metodo non fa altro che richiamare, mediante un ciclo for, cento volte il metodo *test*, con un'attesa di un secondo tra una chiamata e l'altra, memorizzando di volta in volta in un vettore (chiamato *samples*) il valore restituito da *test*, che rappresenta il tempo di esecuzione della query. Il metodo test, dopo aver eseguito la connessione al database tramite *connect*, memorizza in una variabile *t1* l'istante di tempo corrente e subito richiama il metodo *selectquery_to_Vector*, a cui viene passata la stringa seguente per eseguire la ricerca:

select Codfisc from ANAGRAFICA where Cognome = 'Garibaldi9999'

Nell'istruzione successiva si memorizza l'istante corrente in un'altra variabile *t2*. Infine il metodo chiude la connessione al database e restituisce il valore *t2-t1*. Dopo che il ciclo for di *test_manager* è terminato, nel vettore *samples* sono memorizzati i 100 valori misurati. Come ultimi passi, il metodo calcola la media di questi valori e richiama il metodo *write_results* per esportare su file il valore misurato.

Il test è stato ripetuto due volte: la prima in presenza di un indice sul campo Cognome (di tipo *Normal*[7]), la seconda in assenza di indici su tale campo.

[7] Vedere il paragrafo 2.1.1 per ulteriori dettagli.

4.2.2 Risultati ottenuti e considerazioni

Dai test realizzati è stato costruito il grafico riportato in figura 4.2. L'istogramma mette in evidenza ciò che si attendeva, ossia che il tempo di esecuzione della query in presenza di indice sul campo secondario risulta essere minore rispetto a quello ottenuto nel caso in cui l'indice non è presente.

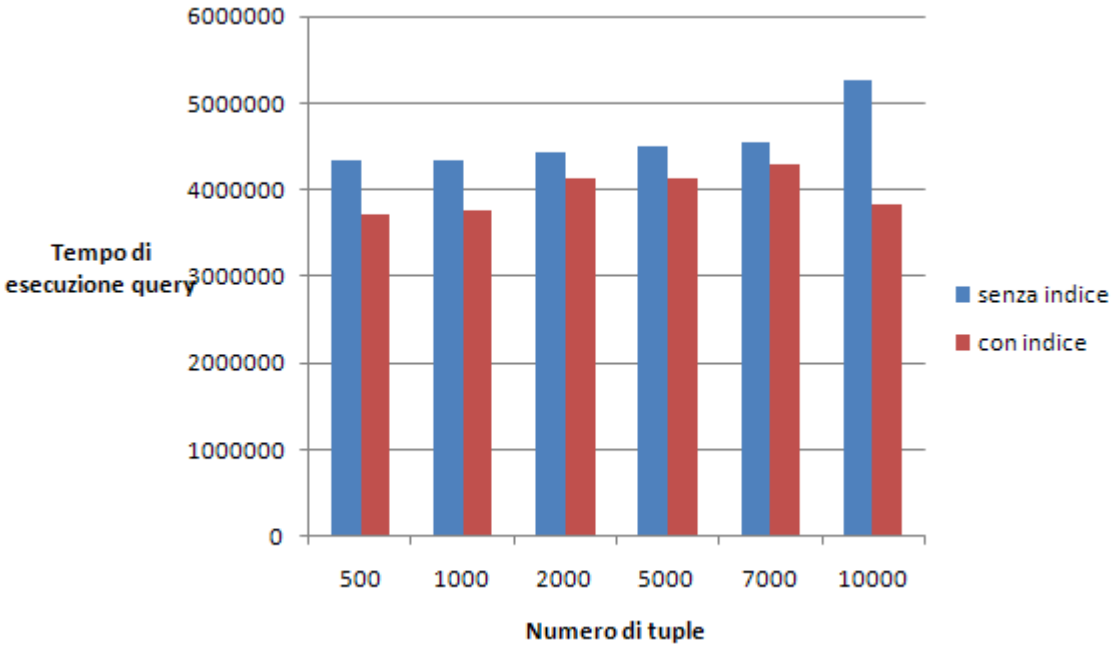


Figura 4.2 Il grafico mostra i tempo di esecuzione della query con e senza indice

4.3 Query di selezione multipla

Lo scopo di questo test è quello di illustrare un caso in cui l'utilizzo di un indice non è vantaggioso. Il caso preso in considerazione è quello di un'interrogazione di ricerca che seleziona un insieme di tuple (differentemente dal caso precedente, in cui veniva selezionata una sola tupla). In questa situazione, infatti, l'utilizzo di un indice non risulta vantaggioso se l'attributo di ricerca è poco selettivo (valore replicato un gran numero di volte nel campo). Ciò è dovuto al fatto che, se le tuple selezionate sono numerose, la scansione delle stesse è paragonabile ad una ricerca lineare. Quando si utilizza l'indice si impiega un numero di accessi ridotto (rispetto a una scansione lineare) per trovare una tupla; tuttavia si deve spendere un ulteriore accesso per recuperare la tupla dalla memoria. Si intuisce allora che, se le tuple selezionate sono tante, il numero di accessi necessari per recuperarle può essere davvero

paragonabile a quello ottenibile nel caso di una scansione lineare. Per illustrare meglio il concetto si considera un esempio banale: supponiamo che il numero di tuple presenti nella tabella sia pari a 10000 e che quelle selezionate siano proprio 10000. Quando si utilizza l'indice sono necessari almeno 10000 accessi per recuperare le tuple dalla memoria, a cui poi vanno aggiunti gli accessi fatti all'indice, mentre una scansione lineare richiederebbe, ovviamente, "solo" i 10000 accessi. In questo caso estremo, quindi, si verifica che il numero di accessi, in presenza di un indice, è addirittura più grande di quello ottenibile in sua assenza.
Il test eseguito, pertanto, ha come obiettivo la misurazione delle differenze nei tempi di esecuzione nel caso di una selezione multipla, al variare della selettività dell'attributo; come nel caso precedente, la ricerca è stata condotta sul campo secondario Cognome. L'idea per fare in modo che la query selezioni un numero di tuple fissato a priori, sia esso pari a *n*, è molto semplice: inserire nella tabella n tuple con lo stesso valore del campo Cognome, ad esempio "Garibaldi0000", mentre nelle altre tutti valori distinti fra loro. In questo modo una ricerca su "Garibaldi0000", evidentemente, restituisce proprio n tuple. Nella realizzazione pratica ciò è stato ottenuto in due passi: dapprima popolando il database con tutte tuple di valore diverso tra loro (per il campo Cognome), e poi in un secondo momento modificando n di esse per inserirvi uno stesso valore ("Garibaldi0000"). Per evitare di compromettere la misura, le n tuple da modificare vengono scelte a caso, come sarà illustrato nel prossimo paragrafo.

4.3.1 Descrizione del caso

Come detto nel paragrafo precedente, la prima cosa da fare è invocare il metodo *popola_db* per riempire la tabella; come per il caso proposto al paragrafo 4.2, le tuple hanno lunghezza fissa e il campo Cognome per ciascuna di esse conterrà valori distinti, costituiti da una parte alfabetica costante e una numerica variabile. In tal caso, però, la dimensione della tabella è fissata ed è pari a 10000 tuple.
Il meccanismo che permette di scegliere e modificare n tuple a caso è contenuto nel metodo *db_update* della classe Test. Innanzitutto, dato che le tuple hanno una parte numerica che varia da 0000 a 9999, per scegliere a caso una tupla da modificare viene generato un numero casuale intero (tramite il metodo *nextInt* della classe *java.util.Random*) tra 0 e 9999. È evidente, però, che il numero così generato non è nella forma voluta, cioè una stringa di 4 cifre. In particolare, detto *x* il numero generato, si osservano i seguenti casi:

1. se $x < 10$, la stringa rappresentante x è composta di un solo carattere;

2. se $10 \leq x < 100$, la stringa rappresentante x è composta di due caratteri;
3. se $100 \leq x < 1000$, la stringa rappresentante x è composta di tre caratteri;
4. solo se $x \geq 1000$, la stringa rappresentante x è composta di 4 caratteri.

L'idea, quindi, è quella di completare con delle cifre di riempimento (ad esempio "0"), laddove necessario (nei casi 1,2,3), la stringa generata in maniera casuale. Operando in questo modo, si seleziona sicuramente una stringa tra quelle presenti nella tabella.[8] Il riempimento viene fatto con delle strutture *if-then* che, a seconda del valore di x, completano la stringa nel modo opportuno. Fatto ciò, si richiama il metodo *update_query* per modificare la tupla corrispondente:

```
update ANAGRAFICA set Cognome = 'Garibaldi0000' where Codfisc = 'FPPGAR2345F"+code+" ' "
```

dove **code** rappresenta la stringa così generata come sopra detto. Per modificare esattamente n tuple non è detto che basti iterare questo procedimento n volte, perché su 10000 valori generati in maniera casuale si possono verificare delle ripetizioni. Per evitare questo problema l'approccio seguito è stato quello di memorizzare i numeri già generati in un vettore, e controllare, all'atto di una nuova generazione, se il numero fosse già stato generato in precedenza; nel caso in cui il valore è già presente, si effettua una nuova generazione.
Dopo aver modificato nel modo descritto il contenuto della tabella, il test effettua le stesse operazioni dell'esempio mostrato al paragrafo 4.2 richiamando i metodi *test_manager* e *test* per eseguire le misure dei tempi di esecuzione nei due casi (presenza e assenza dell'indice sul campo Cognome).

4.3.2 Risultati ottenuti e considerazioni

I risultati ottenuti sono riassunti nella figura 4.3. Il primo grafico riporta i tempi di esecuzione misurati. Come si può notare, il tempo di esecuzione in presenza di indice risulta molto più piccolo rispetto a quello senza indice, quando il numero di tuple selezionate è pari a 100, cioè l'attributo è molto selettivo. Al diminuire della selettività dell'attributo, la differenza tra i tempi di esecuzione è sempre minore, e addirittura per 5001 e 7501 tuple selezionate, il tempo di esecuzione in presenza di indice risulta maggiore rispetto a quello in assenza indice. Questa situazione è apprezzabile ancor meglio osservando il secondo grafico presente in figura 4.3. Tale grafico riporta la variazione percentuale (a parità di tuple selezionate), dei tempi di

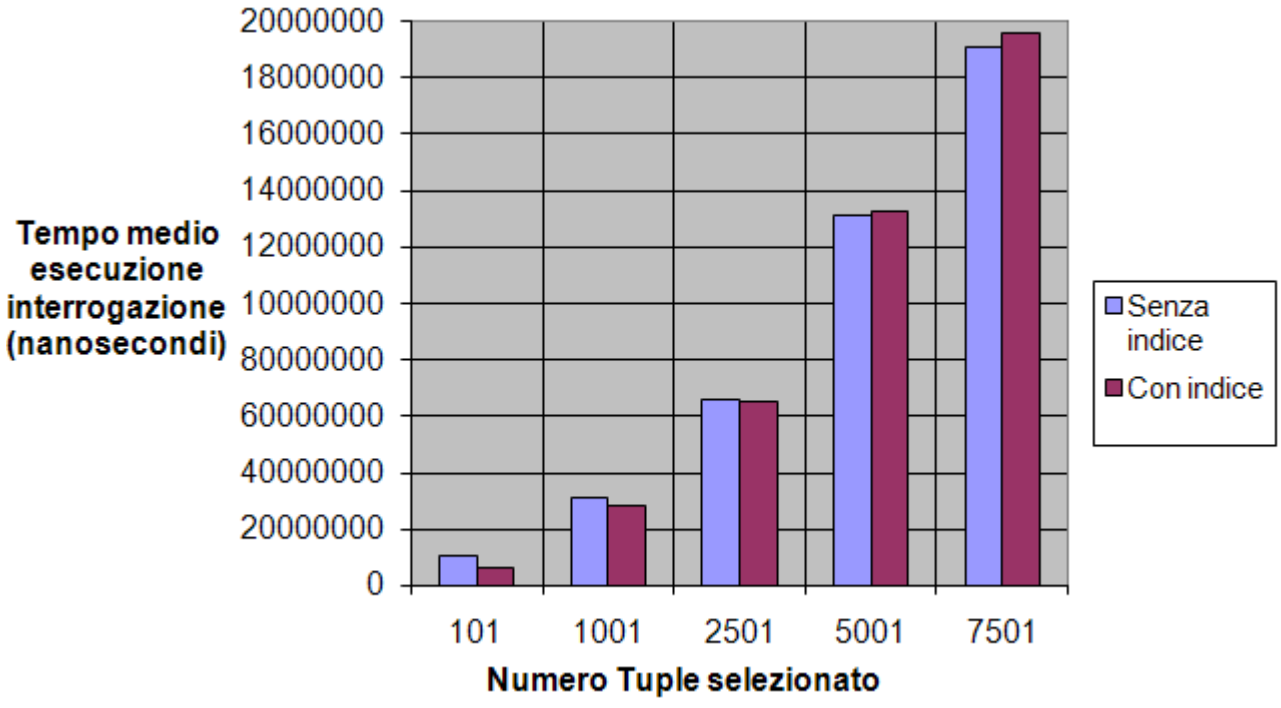
[8] In questo modo non tutte le stringhe tra 0000 e 9999 sono selezionabili, come 1000 ad esempio. Ai nostri fini ciò però non costituisce un problema.

esecuzione delle interrogazioni nei casi di presenza e assenza dell'indice sul campo di ricerca. Detti t_{ind} e t_{noind}, rispettivamente i tempi di esecuzione in presenza e in assenza di indice, la variazione percentuale $\Delta t_{\%}$ viene calcolata secondo la seguente espressione:

$$\Delta t_{\%} = 1 - \frac{t_{ind}}{t_{noind}} \cdot 100$$

Come si può vedere, questo grafico ci permette di apprezzare ancora più chiaramente di quanto effettivamente migliori il tempo di esecuzione in presenza dell'indice. In particolare, si evince che il miglioramento maggiore si ha in corrispondenza di un numero di tuple pari a 101 (miglioramento quasi del 40%). Al crescere del numero di tuple selezionate, il miglioramento diviene sempre più piccolo: esso, infatti, risulta minore del 10% per 1001 tuple selezionate, e minore del 5% per 2501 tuple. Si noti, inoltre, che in corrispondenza dei test a 5001 e 7501 tuple la variazione percentuale diviene negativa. Ciò è dovuto proprio al fatto che quando l'attributo è poco selettivo, l'utilizzo dell'indice non solo non è vantaggioso, ma addirittura grava sulle prestazioni del database, influendo in maniera negativa sui tempi di esecuzione delle interrogazioni.

I risultati ottenuti sono coerenti con quanto detto nel paragrafo 4.3, in quanto si è riscontrato che la presenza dell'indice è vantaggiosa quando il numero di tuple selezionate è basso, ovvero il campo di indicizzazione è molto selettivo, mentre è sempre più deleteria (dal punto di vista delle prestazioni) quando il numero di tuple selezionato è sufficientemente alto.



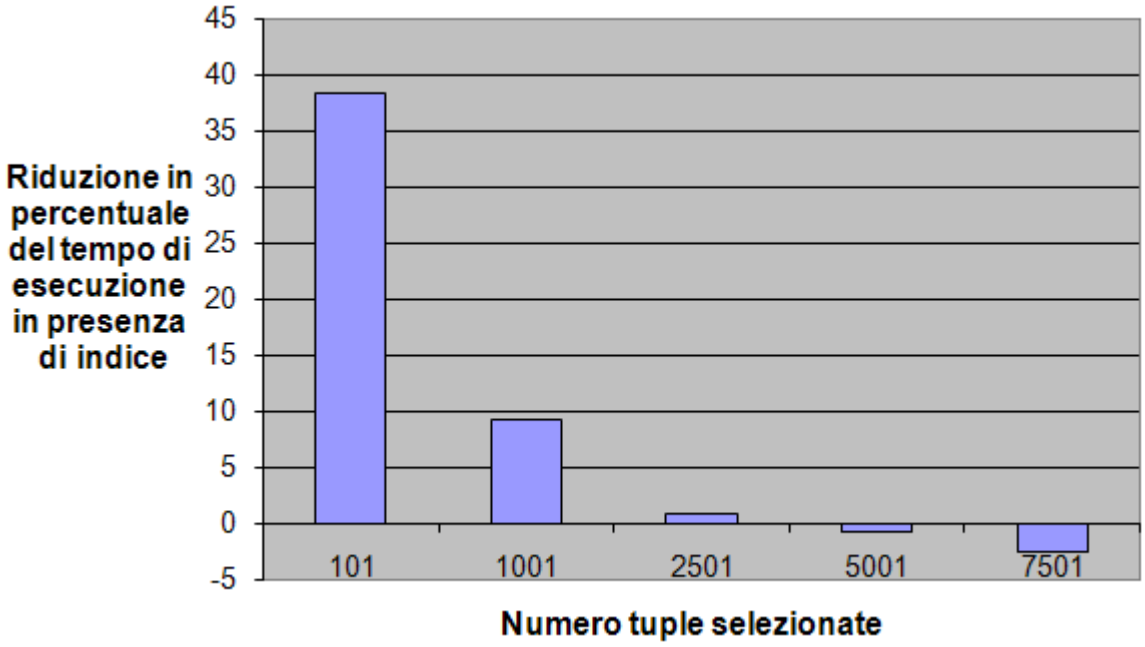


Figura 4.3 Il primo grafico mostra i tempi di esecuzione con e senza indice al variare della selettività dell'attributo; il secondo mostra la variazione in percentuale del tempo di esecuzione tra il caso in cui l'indice è presente e quello in cui è assente.

Bibliografia

[1] Paolo Atzeni, Stefano Ceri, Piero Fraternali, Stefano Paraboschi, Riccardo Torlone: *Basi di dati: Architetture e linee di evoluzione 2/ed,* McGraw-Hill Italia, 2007

http://www.ateneonline.it/atzeni/homeB.asp

[2] S. Navathe, R. Elmasri: *Sistemi di basi di dati – complementi 4/ed,* Pearson-Education Italia, 2005

http://hpe.pearsoned.it/site/show.php?curr_sec=catalogo&sub_sec=cat_sk_libro&ISBN=8871922212

[3] Nicola Vitacolonna: *Cenni di progettazione fisica delle basi di dati 4/ed,* Corso di Basi di dati, Università di Trieste, Giugno 2007

http://users.dimi.uniud.it/~nicola.vitacolonna/download/corsi/bdd/Trieste/2007/dispense/ProgettazioneFisica.pdf

[4] Eric Belden, Bjorn Engsig, Nancy Greenberg, Christopher Jones, Simon Law, Mark Townsend: *Oracle Database Express Edition 2 Day Developer Guide, 10g Release 2 (10.2),* 2006

http://www.oracle.com/pls/xe102/to_pdf?pathname=appdev.102%2Fb25108.pdf&remark=portal+%28Getting+Started%29

[5] The MySQL AB Group: *MySQL 6.0 Reference Manual,* Gennaio 2008

http://downloads.mysql.com/docs/refman-6.0-en.a4.pdf

[6] Kevin Loney: *La guida completa Oracle Database 10g*, McGraw-Hill 2005