Indice

1 Introduzione

Lo scopo della presente dispensa e quello di trattare alcuni argomenti del corso di "Basi di Dati e Multimedia" con l'approfondimento richiesto dal corso stesso.

Più nel dettaglio, si farà riferimento ad alcuni dei metodi efficienti di accesso ai dati: B^+-Tree e strutture Hash per quanto riguarda le basi di dati tradizionali(relazionali) mentre per le basi di dati multimediali si tratteranno k-d-Tree e Grid-File.

La maggior parte delle informazioni per tutti gli argomenti sono tratte da [6] e da [7]. Per i B^+-Tree si è fatto riferimento anche a [1] e [2], per i k-d-Tree a [3] e [5], mentre per i Grid-File a [4].

È sempre bene iniziare ricordando alcune definizioni e concetti che permettono di inquadrare il contesto cui si fa riferimento per facilitare la comprensione degli argomenti esposti.

2 Richiami

Si possono distinguere tre categorie di strutture fisiche per la memorizzazione e l'accesso ai dati:

1. Strutture sequenziali, che a loro volta si dividono in:
   - Seriali.
   - Ad array (le tuple sono di lunghezza fissa).
   - Ordinate.
2. Strutture ad accesso calcolato (hashing).
3. Strutture ad albero.

Per quanto riguarda la prima categoria, composta da strutture usate più per la memorizzazione che per l'accesso, si farà riferimento in particolare alle strutture sequenziali ordinate o *ISAM*(Index Sequential Access Metod). Le caratteristiche principali sono riassumibili come al seguito:

- Si basano su una chiave di ordinamento (può non coincidere con la chiave primaria della tabella).
- Le tuple contenute nella struttura sono ordinate fisicamente secondo l'ordine indotto dalla chiave di ordinamento.

- Una catena di puntatori garantisce la possibilità di scandire le tuple secondo la chiave di ordinamento.

Nel seguito, se non diversamente specificato, si assumerà che i dati siano memorizzati in questo tipo di strutture.

Introduciamo la definizione di *indice* su cui si basano le strutture di accesso ai dati (quelle ad albero e ad accesso calcolato).

Indice 2.1 *È una struttura che migliora le prestazioni di accesso ai dati. Si riferisce ad una chiave di ricerca.*

L'indice si dice *primario* quando la sua chiave di ricerca coincide con la chiave di ordinamento.

La struttura di un indice è costituita da un insieme di record della forma

$$< v, p >$$

dove v è un valore della chiave di ricerca, p è il puntatore alla prima tupla che presenta il valore v della chiave di ricerca. In un indice *secondario* invece, che presenta una chiave di ricerca diversa da quella di ordinamento, p punta ad un bucket[1] di puntatori.

L'indice si dice *denso* se contiene un record per ogni occorrenza della chiave di ricerca che è presente nelle tuple.

L'indice si dice *sparso* se contiene record solo per alcuni valori della chiave di ricerca. Da notare che gli indici secondari sono sempre densi, altrimenti alcune tuple non sarebbero agevolmente accessibili dato che bisognerebbe scorrere al più l'intera pagina in cui sono memorizzate per trovarle.

Le operazioni che solitamente vengono fatte su queste strutture sono:

- **Ricerca** della tupla con valore $\overline{k}$ della chiave (per gli indici secondari occorre leggere anche il bucket di puntatori).

- **Cancellazione** di una tupla con valore $\overline{k}$ della chiave.

- **Inserimento** di una tupla con valore $\overline{k}$ della chiave.

2.1 Estensioni a basi di dati multimediali

Si può estendere quanto detto in precedenza alle basi di dati multimediali, con un grado di complessità maggiore dovuta alla natura dei dati coinvolti (immagini, audio, video, documenti, ...) che in generale sono rappresentati

[1] Un bucket di puntatori è un insieme di puntatori.

con spazi n-dimensionali in cui non si ha piú una singola chiave. Naturalmente questo comporta che, tra l'altro, anche le strutture di accesso ai dati debbano essere pensate per gestire tale multidimensionalità, da cui nasce la necessità di introdurre k-d-Tree e Grid-File: esistono peraltro altre strutture di indicizzazione tra le quali troviamo Point-Quadtree ed R-Tree [9]. In realtà, alcune delle strutture di accesso multi-chiave sono delle versioni generalizzate di quelle a chiave singola ma non sono efficienti nel caso di file altamente dinamici. Uno dei problemi di maggior rilievo è dovuto al fatto che non esiste un ordinamento totale per dati multidimensionali, assunzione su cui si basano invece molte delle strutture di accesso ai dati a chiave singola.

Le interrogazioni (*query*) cui si farà riferimento nella dispensa saranno:

- **Point query:** ricerca di un vettore[2] dato.

- **Range query:** ricerca di tutti i vettori che appartengono ad una data regione.

- **Nearest neighbour query:** ricerca del vettore più vicino ad un vettore dato.

Quest'ultima in particolare è più specifica per basi di dati multimediali: si pensi, per esempio, di cercare in una cartina geografica la città più vicina ad una di riferimento.

Esistono anche altri tipi di query nel contesto multidimensionale, come *partial match query* o *spatial join query* [8].

Si precisa che in questa dispensa ci si riferisce alla stessa cosa parlando di strutture di accesso, di indicizzazione e di ricerca; inoltre sarà trattato il caso particolare in cui i dati siano 2-dimensionali.

3 B$^+$-Tree

Come già accennato i B$^+$-Tree sono degli indici con struttura ad albero caratterizzati da un nodo radice, vari nodi intermedi e vari nodi foglia.

Le caratteristiche della struttura sono le seguenti:

- I nodi dell'albero corrispondono a pagine o blocchi a livello di file system (memoria secondaria).

- I legami tra i nodi dell'albero sono puntatori a pagine.

[2]Per vettore si intende un punto n-dimensionale.

- Ogni nodo, in genere, ha un numero elevato di discendenti (ciò consente di costruire alberi con un numero limitato di livelli nei quali la maggioranza delle pagine è occupata da nodi foglia).
- L'albero è bilanciato, ovvero la lunghezza di ogni cammino che collega la radice ai nodi foglia è costante.
- Inserimenti e cancellazioni non alterano le prestazioni dell'indice nell'accesso ai dati.

3.1 Struttura di un B$^+$-Tree

La struttura di un nodo interno di un B$^+$-Tree di ordine p (*fan-out*) è mostrata in Figura 1, in cui:

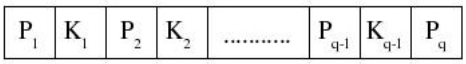

Figura 1: Struttura di un nodo interno.

1. $q \leq p$ ed ogni P_i è un puntatore ad un sotto-albero.
2. In ogni nodo interno, $k_1 < k_2 < \ldots < k_{q-1}$.
3. P_1 punta al sotto-albero che contiene le chiavi $k < k_1$, P_i (con $1 < i < q$) al sotto-albero con chiavi $k_{i-1} \leq k < k_i$ e P_q al sotto-albero con chiavi $k \geq k_{q-1}$.
4. Un nodo interno con q puntatori ha $q - 1$ valori per la chiave.

Ci sono anche dei **vincoli di riempimento per nodi interni** sul numero c di puntatori:

$$\left\lceil \frac{p}{2} \right\rceil \leq c \leq p\,,$$

dove $\lceil \cdot \rceil$ è l'arrotondamento all'intero superiore.

La struttura di un nodo foglia di un B$^+$-Tree di ordine p è mostrata in Figura 2, in cui:

1. $q \leq p$, ogni P_i è un puntatore ai dati e P_{next} punta al prossimo nodo foglia del B$^+$-Tree[3].

[3]Il motivo per cui tali alberi si chiamano B$^+$-Tree e non solamente B-Tree è appunto perchè in questi ultimi le foglie non sono collegate sequenzialmente tra loro.

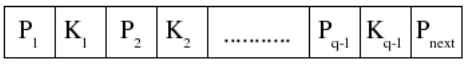

Figura 2: Struttura di un nodo foglia.

2. In ogni nodo foglia, $k_1 < k_2 < \ldots < k_{q-1}$.

3. Ogni P_i è un puntatore ai dati: se l'indice è primario punta alla prima tupla con k_i, se l'indice è secondario al bucket di puntatori.

4. Dati due nodi foglia F_i ed F_j con $i < j$ risulta:

$$\forall K_l \in F_i, \forall K_m \in F_j : k_l < k_m \, .$$

5. Un nodo foglia può contenere fino a $p-1$ valori per la chiave e fino a p puntatori.

6. Tutti i nodi foglia sono allo stesso livello.

Ci sono dei **vincoli di riempimento per nodi foglia** sul numero w di valori per la chiave:

$$\left\lceil \frac{p-1}{2} \right\rceil \leq w \leq p-1 \, .$$

Un esempio di B^+-Tree si può trovare in Figura 3.

3.2 Operazioni su un B^+-Tree

Vediamo ora degli algoritmi ad alto livello per le operazioni di ricerca, inserimento e cancellazione.

Ricerca delle tuple con valore $\overline{k}$ della chiave di ricerca:

- Cercare nel nodo corrente (inizio nodo radice) il più piccolo valore di chiave che sia maggiore di $\overline{k}$.

- Se esiste tale valore (k_i) nel nodo corrente seguire il puntatore P_i, altrimenti seguire P_q.

- Se il nodo raggiunto è un nodo foglia, cercare $\overline{k}$ nel nodo e seguire il corrispondente puntatore verso le tuple, altrimenti tornare al primo passo dell'algoritmo.

Il costo della ricerca inteso come numero di accessi alla memoria secondaria è pari alla profondità dell'albero che si dimostra essere:

$$1 + \log_{\lceil \frac{p}{2} \rceil} \left(\frac{|valori\_chiave|}{\lceil \frac{p-1}{2} \rceil} \right) \quad ,$$

ics214a dove $|\cdot|$ indica la funzione cardinalità.

Una proprietà interessante dei B^+-Tree consiste nel fatto che anche la ricerca di tuple con chiave $k_1 \leq \overline{k} \leq k_2$ (ovvero la range-query) ha costo logaritmico poichè una volta trovato il nodo che contiene k_1 basta seguire i puntatori che collegano i nodi foglia fino a quello contenente k_2[4].

Inserimento di una tupla con chiave $\overline{k}$:

- Ricerca del nodo foglia che dovrebbe contenere $\overline{k}$.
- Se il valore $\overline{k}$ è contenuto nel nodo foglia raggiunto, aggiornare i puntatori verso le tuple, altrimenti inserire $\overline{k}$ nel nodo foglia rispettando l'ordine e creare il puntatore verso la nuova tupla.
- Se non è possibile inserire $\overline{k}$ nel nodo foglia allora va eseguito uno SPLIT del nodo.

SPLIT del nodo (per i nodi interni fare riferimento a quanto riportato tra parentesi):

- Creare due nuovi nodi.
- Inserire i primi $\lceil \frac{p-1}{2} \rceil$ valori ($\lceil \frac{p}{2} \rceil$ puntatori) nel primo nodo.
- Inserire i rimanenti nel secondo.
- Inserire un nuovo puntatore nel nodo padre ed aggiustare i valori in esso contenuti.
- Se il padre contiene già p puntatori $\rightarrow$ SPLIT del padre.

Cancellazione di una tupla con chiave $\overline{k}$:

- Ricerca del nodo foglia che contiene il valore da cancellare.
- Sul nodo foglia raggiunto cancellare il puntatore alla tupla da eliminare.

[4]Se non esistessero i valori k_1 e/o k_2 si considera la chiave più piccola rispettivamente maggiore o minore.

- Se esistono altre tuple con valore $\overline{k}$ della chiave allora fine, altrimenti togliere $\overline{k}$ dal nodo foglia.

- Se il nodo foglia rimane con meno di $\lceil \frac{p-1}{2} \rceil$ chiavi va eseguito un MERGE del nodo.

MERGE di un nodo N:

- Identificare il nodo fratello adiacente da fondere con N.

- Se il nodo individuato contiene un numero di valori chiave (o puntatori nel caso di nodi interni) che sommati a quelli di N rispettano il vincolo di riempimento allora si fondono i due nodi, altrimenti si ridistribuiscono i valori chiave (puntatori) tra gli stessi.

- Si aggiornano i puntatori sul nodo padre.

- Se il padre vìola il vincolo minimo di riempimento $\rightarrow$ MERGE del padre.

La Figura 3 mostra un esempio di inserimento in un B^+-Tree con fan-out= 3. La tupla con chiave G andrebbe inserita nel nodo $\{B, C\}$, ma non si possono più aggiungere chiavi altrimenti si violerebbe il vincolo massimo di riempimento per i nodi foglia; per questo è necessario eseguire uno split del nodo. Come precisato nell'algoritmo di split si creano due nuovi nodi di cui il primo contenente un valore e il secondo i rimanenti: c'è però il problema che il padre non può ospitare un nuovo puntatore altrimenti sarebbe violato il vincolo massimo di riempimento per i nodi interni. L'albero che risulta dopo aver effettuato lo split del padre e aggiornato valori di chiavi e puntatori è mostrato in Figura 4. Da notare che il valore della chiave presente nella radice è il più piccolo del sottoalbero destro come conseguenza della struttura dei nodi.

In Figura 5 si può trovare un esempio di cancellazione in un B^+-Tree con fan-out= 3. Essendo F per ipotesi l'ultima tupla con quel valore della chiave è necessario rimuoverla dal nodo foglia, ma sfortunatamente è violato il vincolo minimo di riempimento per i nodi foglia: bisogna effettuare un merge. Dato che il nodo rimane vuoto, esso viene cancellato e la foglia $\{E\}$ punta ora alla foglia $\{H\}$ mentre il *padre* $\{H\}$ vìola il vincolo minimo di riempimento per i nodi interni. È necessario un merge del *padre* $\{H\}$ con $\{B, E\}$ ma visto che la somma dei puntatori di questi ultimi vìola il vincolo massimo di riempimento sul numero di puntatori si ridistribuiscono i puntatori fra i due nodi. Come conseguenza, la foglia $\{E\}$ cambia padre e la nuova configurazione in Figura 6 mostra l'albero ottenuto dopo la cancellazione.

Da notare che i valori nei nodi interni sono tutti presenti anche nelle foglie.

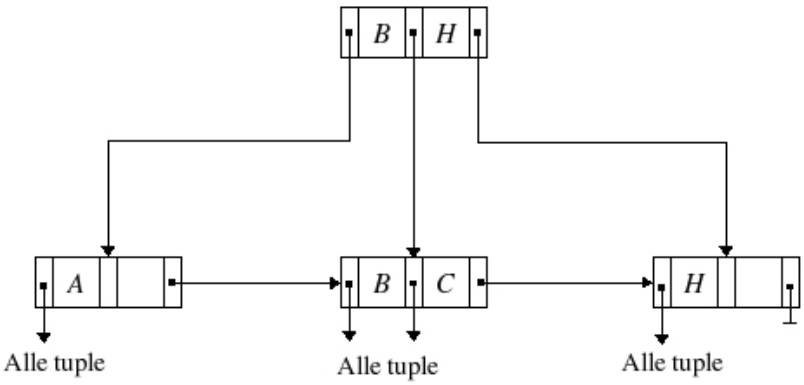

Figura 3: Albero di partenza in cui inserire G.

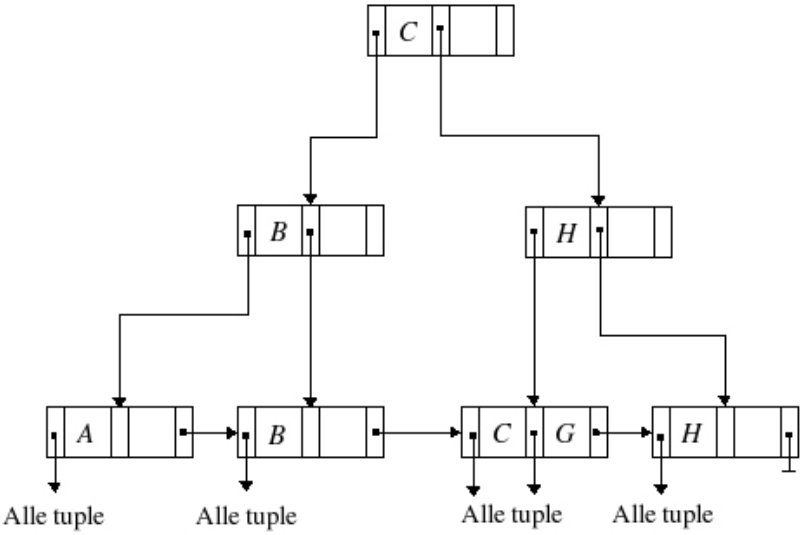

Figura 4: Albero di Figura 3 dopo l'inserimento di G.

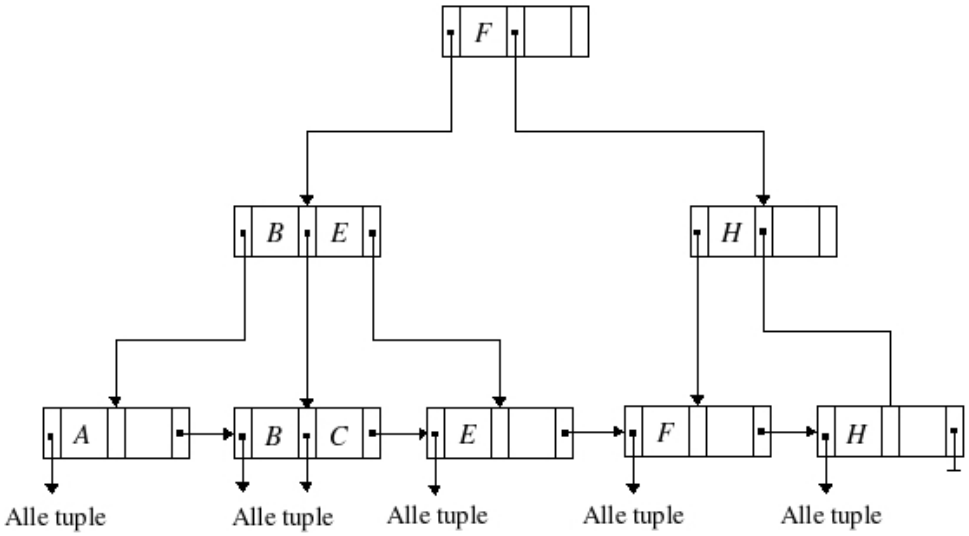

Figura 5: Albero di partenza in cui cancellare F.

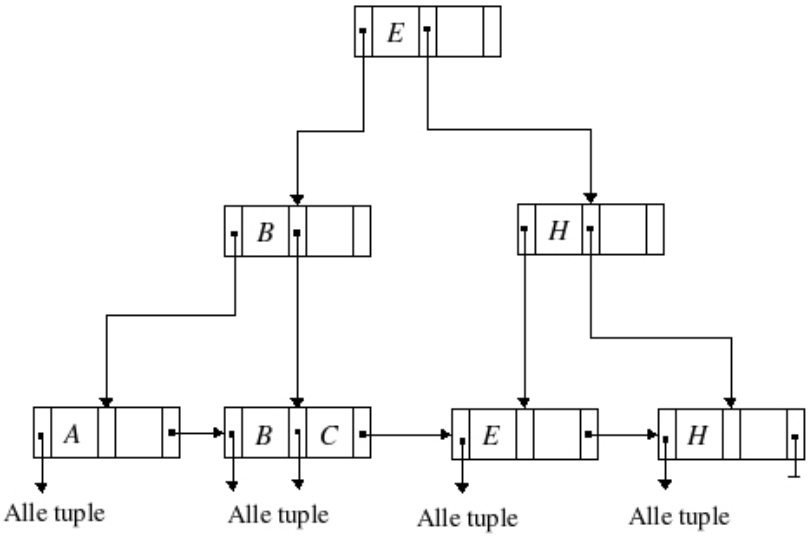

Figura 6: Albero di Figura 5 dopo la cancellazione di F.

4 Strutture ad accesso calcolato (Hash)

In questo tipo di strutture si adotta una tecnica diversa rispetto ai B^+-Tree nel senso che non si memorizza più la coppia $< v, p >$ ma viene utilizzata una funzione di *hashing* per mappare i valori della chiave di ricerca dell'indice sugli indirizzi di memoria secondaria (pagine dati). L'obiettivo è di ridurre il numero di accessi alla memoria di massa.

Per la costruzione di una struttura ad accesso calcolato è necessario:

1. Stimare il numero di valori chiave che saranno contenuti in media nell'indice.

2. Allocare in memoria secondaria n bucket di puntatori (cioè pagine che contengono puntatori): $B = \{B_1, B_2, \ldots, B_n\}$.

3. Definire una funzione di *folding* che trasforma i valori chiave in interi positivi:
$$F: \quad k \longmapsto \mathbb{Z}^+.$$

4. Definire una funzione di *hashing* che a partire da un intero positivo calcola il corrispondente bucket.
$$H: \quad \mathbb{Z}^+ \longmapsto B.$$

La Figura 7 mostra un esempio di indicizzazione secondaria con funzione di hashing; notare la presenza del conflitto delle chiavi di ricerca "Rossi" e "Verdi", che vengono mandate entrambe nel bucket 4 dalla funzione H.

Vediamo i passi da seguire per la **ricerca** di una tupla con valore $\overline{k}$ della chiave:

- Applicare la funzione di hashing a $\overline{k}$: $H(F(\overline{k}))$.

- Leggere il bucket che corrisponde a $H(F(\overline{k}))$[5].

- Scandire i puntatori del bucket per trovare le tuple.

Una buona funzione di hashing deve distribuire i valori in modo casuale allo scopo di diminuire la probabilità di *collisione* che si verifica quando dati due valori della chiave k_1 e k_2 con $k_1 \neq k_2$ risulta:
$$H(F(k_1)) \; = \; H(F(k_2)).$$

[5]Questo passo, assieme al precedente, comporta la lettura di una sola pagina indice, non dipende dal numero di valori in essa contenuti.

CHIAVE RICERCA	$H(F())$
Bianchi	6
Gialli	7
Neri	3
Rossi	4
Verdi	4

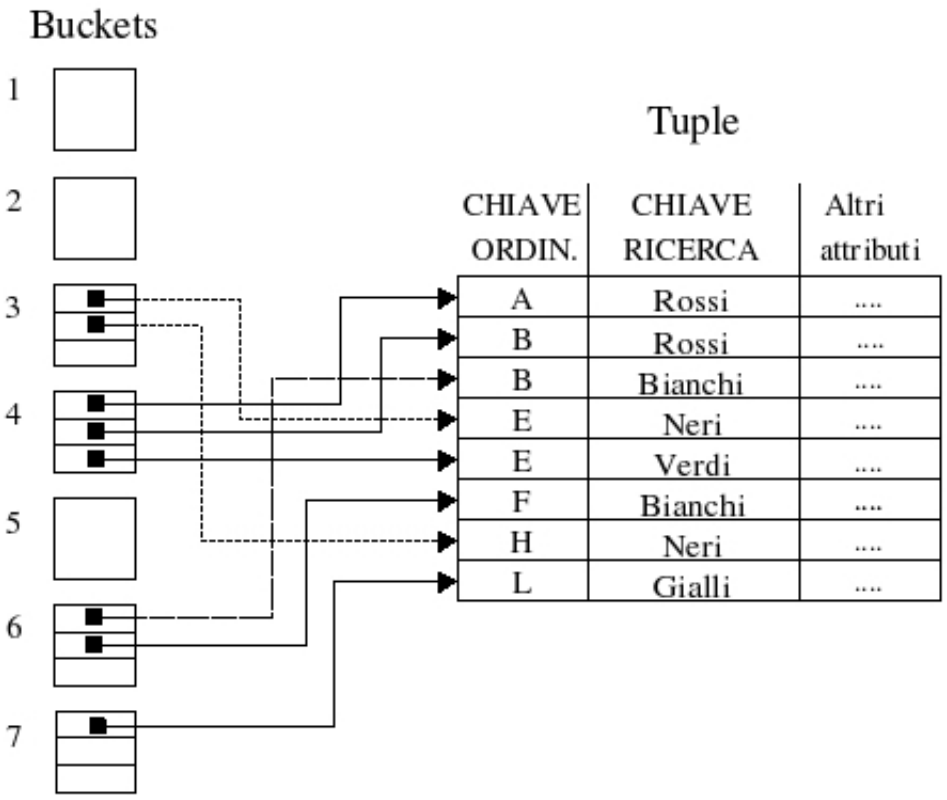

Figura 7: Esempio di indicizzazione con funzione di hashing.

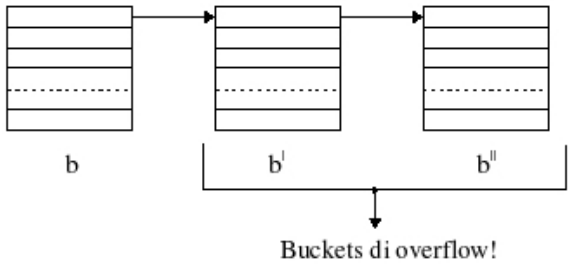

Figura 8: Catena di overflow.

Le collisioni hanno conseguenze negative quando un bucket è pieno non potendo più accogliere nuovi puntatori: si genera a questo punto un bucket di overflow collegato al precedente. Questa tecnica, come mostrato in Figura 8, viene replicata anche in caso di completamento dello spazio disponibile nel nuovo blocco, dando luogo a *catene di overflow* che peggiorano le prestazioni di accesso ai dati visto che viene a mancare la caratteristica fondamentale di dover leggere da memoria secondaria una sola pagina per l'indice.

La probabilità che un bucket riceva t chiavi su n inserimenti è data dalla seguente distribuzione binomiale:

$$p(t) \;=\; C_{n,t} \cdot \left(\frac{1}{|B|}\right)^{t} \cdot \left(1 - \frac{1}{|B|}\right)^{n-t},$$

dove $|B|$ indica la cardinalità dell'insieme dei bucket B e $C_{n,t}$ è il coefficiente binomiale[6]. Come accenno di dimostrazione si fa notare che il primo termine rende conto del numero di modi diversi in cui si possono scegliere t tuple da un insieme di n, $1/|B|$ è la probabilità che la funzione di hash produca un determinato valore che identifica un bucket, mentre $(1/|B|)^t(1-1/|B|)^{n-t}$ è la probabilità di ottenere, in n prove (inserimenti), una particolare sequenza di t successi (chiavi) ed $n-t$ insuccessi.

In seguito a quanto visto, possiamo calcolare la probabilità p di avere più collisioni di quanti siano i puntatori disponibili nel bucket, perchè è in questo caso che ha origine la catena di overflow:

$$p \;=\; 1 - \sum_{i=0}^{F} p(i),$$

con F numero di puntatori che può ospitare il bucket e $p(i)$ la distribuzione precedente. La lunghezza delle catene di overflow decresce all'aumentare di

[6] $C_{n,t} = \binom{n}{t} = \frac{n!}{t!(n-t)!}$.

F perchè le probabilità di collisione elevata si ammortizzano su uno spazio più grande.

In assenza di collisioni, quindi, questo metodo di accesso consente di eseguire ricerche puntuali di tuple con una sola operazione di ingesso/uscita per localizzare il blocco interessato: la presenza di collisioni tuttavia non porta in media a peggioramenti significativi permettendo di dire che il costo della ricerca di tuple con chiave $\overline{k}$ è costante e di circa una pagina.

Nel caso di accessi basati su intervalli non si ottiene un beneficio analogo a quello degli accessi puntuali, perchè le funzioni hash tendono a sparpagliare valori anche vicini nel dominio applicativo su blocchi diversi: il costo della ricerca di tuple con chiave $k_1 \leq \overline{k} \leq k_2$ dipende dunque dall'intervallo $[k_1, k_2]$.

5 k-d-Tree

Il k-d-Tree[7] è un albero *binario* di ricerca multidimensionale in cui ogni nodo indicizza un punto in k dimensioni.

Tratteremo dapprima il caso in cui $k = 2$, tanto per non scoraggiare subito la lettura, per poi discutere l'estensione al caso più generico.

Una definizione che è bene dare ora è quella di *livello di un nodo* di un albero:

Livello di un nodo 5.1 *Sia T la radice dell'albero ed N un generico nodo, il livello di N è una funzione definita ricorsivamente come segue:*

$$\textbf{level}: \quad N \longmapsto \mathbb{Z}^+,$$

$$\textbf{level}(N) = \begin{cases} 0 & \text{se } N \equiv T, \\ \textbf{level}(P) + 1 & \text{se } P \text{ è padre di } N. \end{cases}$$

5.1 Struttura di un 2-d-Tree

Ogni nodo di un 2-d-Tree può essere pensato come a un *record*[8] con la seguente struttura:

[7] k-d-Tree sta per "k dimensional tree".

[8] Un record è un'aggregazione di variabili di tipo diverso.

```
RECORD Nodetype :
{
    INFOTYPE    :  info;
    REAL        :  xval;
    REAL        :  yval;
    ↑Nodetype   :  llink;
    ↑Nodetype   :  rlink;
}
```

dove *info* sono informazioni relative al nodo che dipendono dall'applicazione, *xval* e *yval* sono gli *attributi*(su cui è definito un ordinamento) che identificano un punto nel piano, *llink* e *rlink* sono puntatori rispettivamente al figlio sinistro e destro.

I vincoli strutturali impongono che ogni livello dell'albero discrimini per un attributo e, più nel dettaglio, devono valere le seguenti condizioni:

- Sia N un generico nodo tale che **level**(N) sia pari, allora:
  - Per ogni nodo M appartenente al sottoalbero sinistro di N, si ha che $M.xval < N.xval$.
  - Per ogni nodo M appartenente al sottoalbero destro di N, si ha che $M.xval \geq N.xval$.
- Sia N un generico nodo tale che **level**(N) sia dispari, allora:
  - Per ogni nodo M appartenente al sottoalbero sinistro di N, si ha che $M.yval < N.yval$.
  - Per ogni nodo M appartenente al sottoalbero destro di N, si ha che $M.yval \geq N.yval$.

Ad ogni punto si può pensare che sia associata una regione rettangolare, la quale viene implicitamente divisa in due parti uguali tramite una linea verticale oppure orizzontale passante per il punto stesso in dipendenza dell'attributo discriminante cioè a seconda del fatto che il nodo sia ad un livello pari o dispari rispettivamente. In altre parole, è il livello di un nodo che determina la *splitting dimension*, ovvero la dimensione lungo la quale un punto divide la regione a lui associata. Tanto per ribadire nuovamente i vincoli imposti dalla struttura, si fa notare che:

- se la linea di divisione di una regione ad opera di un punto è verticale, tutti gli elementi del sottoalbero destro cadono nella sotto-regione di destra o sulla linea divisoria mentre tutti gli elementi del sottoalbero sinistro cadono nella sotto-regione di sinistra.

- se la linea di divisione di una regione è orizzontale, invece, tutti gli elementi del sottoalbero destro cadono nella sotto-regione di sopra o sulla linea divisoria mentre tutti gli elementi del sottoalbero sinistro cadono nella sotto-regione di sotto.

Alla radice dell'albero è associata l'intera regione di partenza.

La Figura 9 mostra un esempio di 2-d-Tree in cui i punti sono inseriti nell'ordine con cui si presentano nella seguente tabella, la quale raccoglie dati di interesse su alcune osservazioni di una lastra radiografica.

Diagnosi/Osservazione	$(\mathbf{xval}, \mathbf{yval})$
Cartilagine	$(19, 45)$
Frattura	$(48, 50)$
Osso	$(38, 30)$
Ombra	$(54, 40)$
Rumore	$(4, 4)$

Osservando la figura si può facilmente intuire, anche se in seguito verrà definita formalmente, la regione associata ad ogni punto. Le linee tratteggiate sono quelle da poco menzionate che definiscono la partizione delle regioni a seconda della *splitting dimension*. I nodi foglia, non avendo discendenti, non bisecano la propria regione.

Questi alberi non sono bilanciati in generale e, nel caso peggiore, possono degenerare in una lista. Per ovviare a questa limitazione i punti dovrebbero essere inseriti casualmente in modo che il valore atteso della profondità dell'albero sia logaritmico, oppure si potrebbero applicare delle tecniche di ri-bilanciamento per mantenere il tempo di ricerca logaritmico.

5.2 Operazioni su un 2-d-Tree

Vengono presentati degli algoritmi ad alto livello per la ricerca, della quale ne verranno esaminati tre tipi, per poi passare a inserimento e cancellazione.

Point query, ovvero ricerca di un punto $\overline{P} = (x, y)$:

- Sia Q inizialmente il nodo radice che si assume essere di livello pari.
- Se $(Q.xval, Q.yval) = \overline{P}$ il punto è stato trovato e fine, altrimenti:
- se Q è di livello pari:
    - se $x < Q.xval$ allora $Q = Q.llink$, altrimenti $Q = Q.rlink$.
- se Q è di livello dispari:

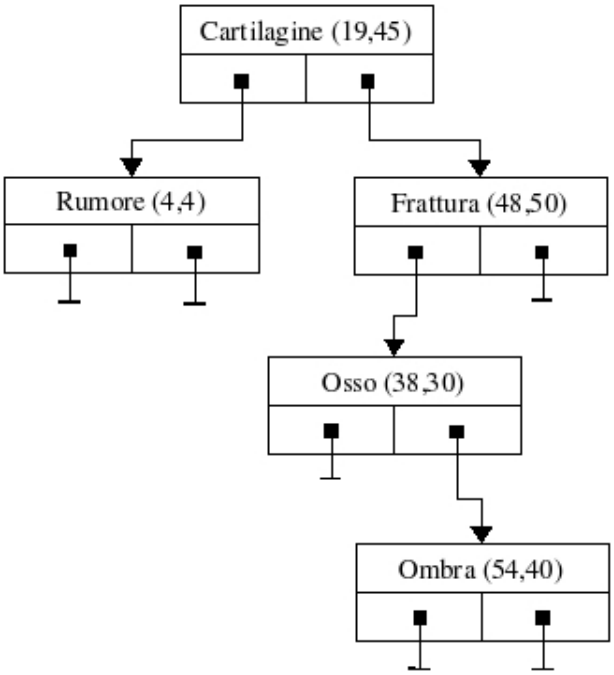

Figura 9: Esempio di 2-d-Tree.

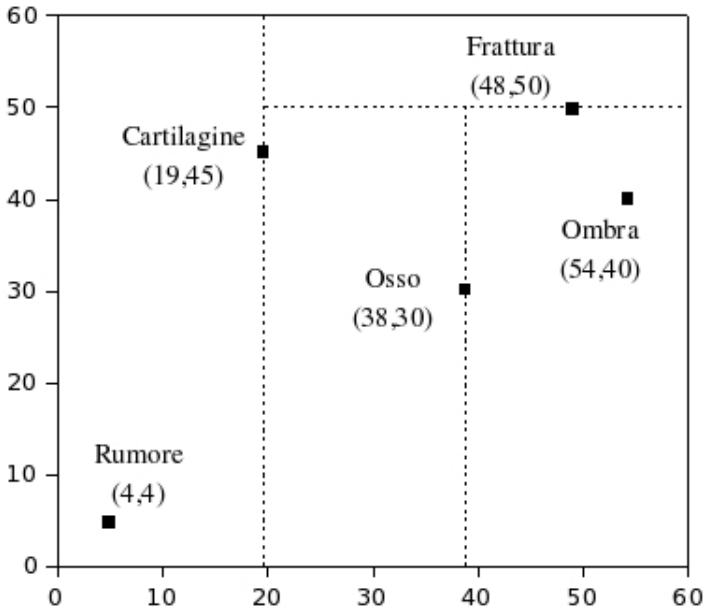

Figura 10: Partizione della regione $[0, 60] \times [0, 60]$ ad opera del 2-d-Tree di Figura 9.

- se $y < Q.yval$ allora $Q = Q.llink$, altrimenti $Q = Q.rlink$.

- Si riparte dal secondo passo fino a che non si trova $\overline{P}$ o si raggiunge la fine dell'albero (puntatori a NULL).

Il tempo di ricerca di $\overline{P}$ è logaritmico nel numero totale di nodi nel caso di un albero bilanciato, tuttavia, come già accennato, potrebbe anche diventare lineare nel caso in cui il k-d-Tree sia degenerato in una lista.

Range query, ovvero ricerca di tutti i punti P_i che appartengono ad una regione passata in ingresso che assumiamo essere rettangolare o circolare, come si mostra la Figura 11. Invece di calcolare di volta in volta la regione rettangolare associata ad un punto, è conveniente memorizzare questa informazione in ogni singolo nodo. La nuova struttura è quindi un record come il precedente con l'aggiunta di quattro campi, come si vede di seguito:

```
RECORD Nodetype :
{
    INFOTYPE   :  info;
    REAL       :  xval;
    REAL       :  yval;
    ↑Nodetype  :  llink;
    ↑Nodetype  :  rlink;
    REAL       :  xlb, xub;
    REAL       :  ylb, yub;
}
```

dove i suffissi "lb" ed "ub" stanno a significare rispettivamente *lower bound* e *upper bound*. Si intuisce che i campi identificano un rettangolo nel piano specificato da due coordinate: il vertice in basso a sinistra (xlb, ylb) e quello in alto a destra (xub, yub) o, equivalentemente, la regione $[xlb, xub] \times [ylb, yub]$.

Vediamo ora quale è effettivamente la regione associata ad un nodo N con padre P:

- se **level**(P) è pari:

$$\left.\begin{array}{rcl} N.xlb & = & P.xlb \\ N.xub & = & P.xval \end{array}\right\} \quad \text{se } N = P.llink.$$

$$\left.\begin{array}{rcl} N.xlb & = & P.xval \\ N.xub & = & P.xub \end{array}\right\} \quad \text{se } N = P.rlink.$$

$$\begin{array}{rcl} N.ylb & = & P.ylb \\ N.yub & = & P.yub \end{array}$$

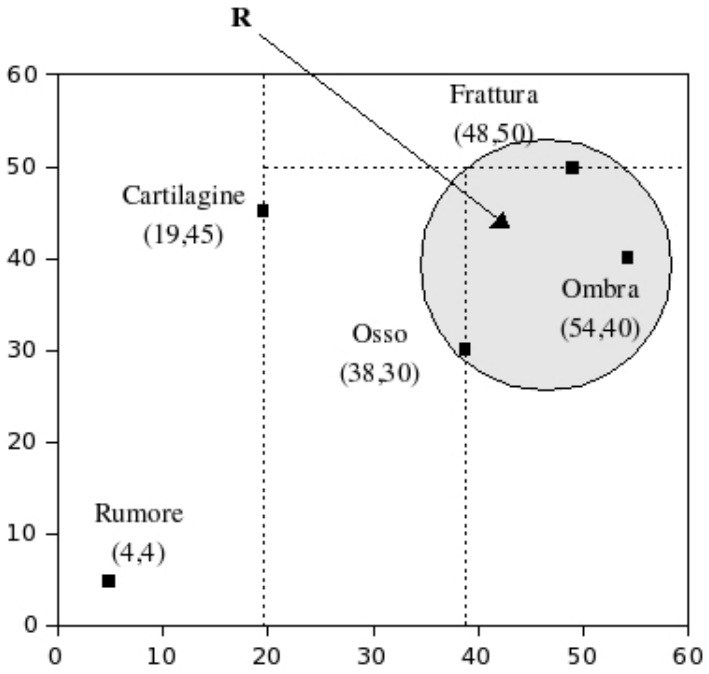

Figura 11: R è la regione di cui si vogliono trovare i punti contenuti.

- se **level**(P) è dispari, simmetricamente:

$$\left.\begin{array}{rcl} N.ylb & = & P.ylb \\ N.yub & = & P.yval \end{array}\right\} \quad \text{se } N = P.llink.$$

$$\left.\begin{array}{rcl} N.ylb & = & P.yval \\ N.yub & = & P.yub \end{array}\right\} \quad \text{se } N = P.rlink.$$

$$\begin{array}{rcl} N.xlb & = & P.xlb \\ N.xub & = & P.xub \end{array}$$

In realtà, come conseguenza dei vincoli di costruzione, la regione di un punto può essere aperta a destra e/o superiormente. Di sicuro, la regione del figlio sinistro di un nodo di livello pari è aperta almeno a destra mentre quella del figlio di un nodo di livello dispari è aperta almeno superiormente; bisogna vedere poi quali sono i vincoli ereditati dal padre. La tabella seguente mostra le regioni effettivamente assegnate ai punti del 2-d-Tree di Figura 9:

Diagnosi/Osservazione	**Regione associata**
Cartilagine	$[0, 60] \times [0, 60]$
Frattura	$[19, 60] \times [0, 60]$
Osso	$[19, 60] \times [0, 50)$
Ombra	$[38, 60] \times [0, 50)$
Rumore	$[0, 19) \times [0, 60]$

Da notare che al nodo radice è associata la regione che contiene tutti i punti della specifica applicazione.

Dovrebbe essere chiaro che una certa regione non è associata in modo assoluto al punto, cioè, se dovessimo costruire due k-d-Tree con gli stessi punti ma con un diverso ordine di inserimento, le regioni di ogni nodo (tranne che per la radice) dopo i due inserimenti non coinciderebbero.

Vediamo ora un possibile algoritmo per la *range query*:

- Sia R la regione considerata per la query, N un generico nodo e $reg(N)$ la regione a lui associata. Si procede ricorsivamente partendo dalla radice e attraversando l'albero:
- se N è una foglia e $(N.xval, N.yval) \in R$, allora ritorna $(N.xval, N.yval)$;
- se N è un nodo interno allora:
  - se $reg(N) \subseteq R$ ritorna tutti i punti dei sottoalberi di N compreso $(N.xval, N.yval)$,
  - se $reg(N) \cap R \neq \phi$: se $(N.xval, N.yval) \in R$ ritorna $(N.xval, N.yval)$ e visita i figli di N che intersecano R,
  - se $reg(N) \cap R = \phi$ la ricerca termina.

Dovrebbe essere chiaro che la complessità dell'algoritmo diminuisce al diminuire del numero di regioni intersecate da R, poichè si riducono il numero di esplorazioni ricorsive.

Un'osservazione importante che permette di risparmiare costo computazionale consiste nel fatto che se una regione associata ad un punto non interseca R, allora non lo fanno neanche quelle associate ai nodi discendenti dato che, in generale, la regione associata ad un figlio è contenuta in quella del padre.

Si può dimostrare che la complessità computazionale della range-query è $O(l + \sqrt{n})$, dove l è il numero di punti che cadono in R.

Nearest neighbour query, ovvero, dato un punto $\overline{P} = (x, y)$ detto *target point* trovare il punto $\overline{Q} = (xval^\star, yval^\star)$ più vicino[9].

[9]Più vicino secondo la distanza Euclidea.

L'algoritmo di forza bruta richiederebbe un tempo lineare nel numero di punti, dato che calcolando la distanza di ognuno di essi da $\overline{P}$, troveremmo sicuramente quello più vicino. La complessità diventa comunque logaritmica se si assume una certa distribuzione dei punti e del target.

Vediamo uno pseudo-algoritmo che sarà presentato in maniera più discorsiva rispetto alle volte precedenti.

Si trova, come prima approssimazione, il nodo foglia F cui è associata la regione che contiene $\overline{P}$, il quale è marcato con "$\times$" in Figura 12: il punto $(F.xval, F.yval)$ non è necessariamente il vicino più vicino ma almeno sappiamo che esso, potenzialmente, si troverà nella circonferenza centrata in $\times$ e raggio la distanza tra $\overline{P}$ ed F. A questo punto si considera il nodo padre di F per vedere se nell'altro figlio esiste un vicino più vicino a quello finora trovato; nell'esempio di Figura 12, questo non è possibile perchè la regione delimitata dalla circonferenza non interseca l'area accupata dal fratello di F (Figura 13). Se non può esistere una soluzione migliore nell'altro figlio, si sale di un livello, altrimenti si aggiorna eventualmente la soluzione prima di salire comunque di un livello per esplorare ricorsivamente le regioni di intersezione. Con riferimento all'esempio, dopo la regione del fratello di F, si vanno ad esaminare le due regioni a sinistra (l'ordine è ininfluente) e in quella superiore si trova il vicino più vicino.

Dalla discussione precedente si sarebbe dovuto riconoscere che, una volta trovata la prima approssimazione, l'algoritmo segue sostanzialmente lo stesso principio di una *range-query* con la differenza che questa volta la regione R viene raffinata man mano che si trova un canditato più vicino e i punti che cadono in R, inoltre, non vengono restituiti tutti ma solo il più vicino al *target*.

Inserimento del nodo relativo ad un punto $\overline{P} = (xval^\star, yval^\star)$:

- Ricerca del nodo foglia che dovrebbe contenere $\overline{P}$.
- Creazione del nodo e aggiornamento dei campi del record relativo al punto $\overline{P}$.

Anche se non è stato detto in modo esplicito, è chiaro che i puntatori ai figli presenti nelle foglie vanno messi a *NULL*.

Ora, si può anche definire in altro modo la regione associata ad un punto, che è quella più piccola cui esso appartiene dopo l'inserimento.

La *creazione* di un k-d-Tree consiste nella creazione del nodo radice con relativo aggiornamento dei campi statici e puntatori a *NULL*, seguita dall'inserimento dei nodi successivi. Con una procedura ricorsiva si costruisce un k-d-Tree in $O(n \log n)$, dove n è il numero di punti.

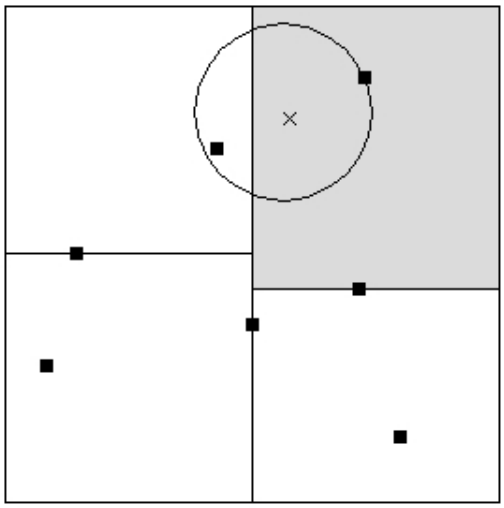

Figura 12: La regione evidenziata contiene il target.

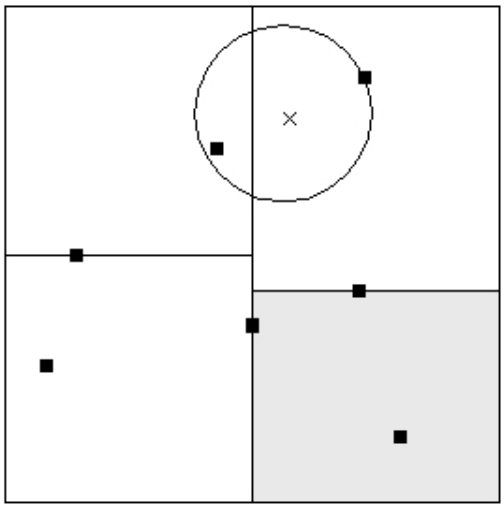

Figura 13: La regione evidenziata è la prima con cui viene fatto il test di intersezione.

Cancellazione del nodo relativo ad un punto $\overline{P} = (xval^\star, yval^\star)$: la cancellazione di un nodo foglia non porta ad alcun problema poichè basta assegnare a $NULL$ il puntatore del padre. Se il nodo da cancellare è interno, invece, bisogna trovare una riorganizzazione dell'albero in modo tale che siano nuovamente rispettati i vincoli strutturali.

La prima soluzione consiste nel cancellare il nodo e reinserire tutti quelli dei suoi sottoalberi ma è troppo oneroso computazionalmente; la seconda soluzione, simile alla precedente, consiste in una cancellazione logica, nel senso che il nodo da rimuovere viene marcato come "eliminato" in modo che non sia preso in considerazione nelle successive ricerche: quando l'albero dovrà essere ricostruito tutti i nodi marcati saranno rimossi. Con la terza soluzione, quella più usata, si cerca nei sottoalberi un candidato che possa essere sostituito al posto del nodo interno da eliminare. A questo punto, dopo la sostituzione, andrà cercato ricorsivamente il candidato del candidato e così via fino a che la ricerca si propagherà ad un nodo foglia, che verrà eliminato facendo terminare la ricorsione.

Più nel dettaglio:

- Cercare il punto da eliminare: sia esso associato al nodo N.
- Se N è una foglia, assegnare a $NULL$ il puntatore del padre.
- Se N è interno:
    - Cercare un nodo candidato R nel sottoalbero destro o sinistro di N.
    - Rimpiazzare i campi statici (quelli non link) di N con quelli di R.
    - Cancellare R.

Come trovare il candidato giusto? Sia N il nodo interno da cancellare:

- se $\mathbf{level}(N)$ è pari:
    - se esiste il sottoalbero destro di N il candidato è il nodo R tale che $R.xval$ è minore o uguale del corrispondente campo in tutti i nodi del sottoalbero destro,
    - se invece $N.rlink = NULL$, si scambiano i puntatori $llink$ ed $rlink$ di N, in modo tale che quello che prima era il sottoalbero sinistro di N ora sia quello destro e viceversa; si ritorna al passo precedente.
- se $\mathbf{level}(N)$ è dispari:
    - se esiste il sottoalbero destro di N il candidato è il nodo R tale che $R.yval$ è minore o uguale del corrispondente campo in tutti i nodi del sottoalbero destro,

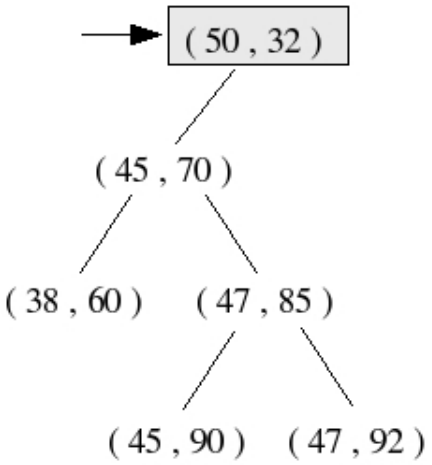

Figura 14: k-d-Tree da cui si vuole cancellare il nodo $(50, 32)$.

- se invece $N.rlink = NULL$, si scambiano i puntatori $llink$ ed $rlink$ di N, in modo tale che quello che prima era il sottoalbero sinistro di N ora sia quello destro e viceversa; si ritorna al passo precedente.

Come esempio, la Figura 14 mostra un k-d-Tree da cui si vuole cancellare il nodo $(50, 32)$ di livello pari; non essendo presente il sottoalbero destro, si scambiano i puntatori ai sottoalberi e al posto della radice si sostituisce il nodo con $xval$ più piccolo: il risultato della cancellazione è quello di Figura 15.

Le osservazioni che seguono si riferiscono all'attributo $xval$ ma la stesso vale anche per $yval$. Il primo passo della ricerca del candidato è facilmente giustificabile se si guardano i vincoli strutturali, il secondo passo richiede invece qualche spiegazione in più. Ci si potrebbe chiedere per quale motivo, se non esistesse il sottoalbero destro, non si potrebbe cercare in quello sinistro il candidato con $xval$ maggiore, di modo che, sostituendolo ad N, rimangano ancora validi i vincoli di costruzione. Il problema nasce dal fatto che l'inserimento non è simmetrico perchè da una parte si ha la condizione maggiore-uguale e dall'altra la condizione strettamente minore; se nel sottoalbero sinistro esistessero almeno due candidati e si procedesse con il ripiazzamento di N, il k-d-Tree ottenuto non sarebbe lecito poichè per almeno un nodo M appartenente al sottoalbero sinistro di N, avremmo che $M.xval = N.xval$ (Figura 14). Procedendo come specificato nel secondo passo dell'algoritmo di ricerca del candidato, invece, ci si riporta al caso di esistenza del sottoalbero destro di N.

La complessità della cancellazione di un nodo nel caso di inserimento favorevole è logaritmica.

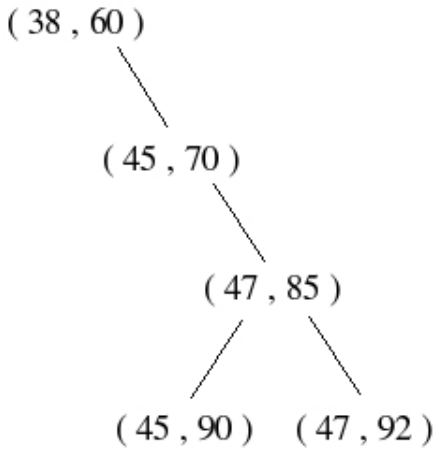

Figura 15: k-d-Tree di Figura 14 dopo la cancellazione del nodo $(50, 32)$.

5.3 Struttura di un k-d-Tree

Estendendo quanto visto al caso generale in k dimensioni, si devono fare innanzitutto alcune modifiche alla struttura del nodo. Non ci sono più solamente due attributi ma, in generale, un vettore di k attributi reali:

REAL : value[k].

I campi che rendevano conto della regione associata ad un nodo dovranno ora essere adeguati a rappresentare spazi multidimensionali.

L'attributo discriminante i sul quale fare i confronti è ora calcolato in questo modo:

$$i = \mathbf{level}(N) \,\mathbf{mod}\, k\,,$$

dove **mod** è il resto della divisione intera.

Supponendo che N sia un generico nodo tale che $\mathbf{level}(N) \,\mathbf{mod}\, k = i$, i vincoli strutturali sono:

- Per ogni nodo M appartenente al sottoalbero sinistro di N, si ha che $M.value[i] < N.value[i]$.

- Per ogni nodo M appartenente al sottoalbero destro di N, si ha che $M.value[i] \geq N.value[i]$.

Il pricipio con cui vengono effettuate le operazioni è come quello in due dimensioni.

6 Grid-File

I Grid-File sono una tecnica di indicizzazione basata su una partizione a griglia (*grid-partition*) dello spazio di ricerca, le cui regioni indotte vengono organizzate opportunamente in *cluster*[10]. Lo scopo è di riuscire a recuperare un singolo *record* (punto nello spazio) con al più due accessi a disco:

- il primo accesso per la corretta porzione della directory, la quale, come vedremo, prende il nome di *grid-array*;

- il secondo per il cluster che contiene il dato.

Spazio di ricerca e spazio dei record sono usati come sinonimi.

Nelle tecniche di ricerca che utilizzano gli alberi, il confine tra due differenti regioni dello spazio di ricerca è determinato direttamente dai valori contenuti nella struttura e quindi cambia continuamente durante il tempo di vita del file dinamico su cui è costruito l'albero. Con una tecnica come quella usata dai Grid-File, invece, i contorni delle regioni si trovano in posizioni non dipendenti dai singoli record rispetto al contenuto del file: l'adattamento alla variabilità di quest'ultimo viene fatto attivando o disattivando alcuni contorni, come si capirà meglio nel seguito. L'utilità delle differenti tecniche di partizionamento dipendono dalla distribuzione dei dati: quella a griglia si può adottare nel caso in cui gli attributi dei record siano indipendenti, l'altra nel caso di distribuzione uniforme dei record.

Nella trattazione dei Grid-File si fanno generalmente alcune assunzioni:

1. Il numero di cluster è illimitato.

2. Ogni cluster ha capacità di c record, con $10 \leq c \leq 1000$: nel caso in cui la capacità dei record sia meno di 10 o più di 1000 saranno utilizzate altre strutture dati.

3. Le differenze temporali impiegate per accedere a cluster diversi sono ignorate, in modo che sia possibile misurare il tempo richiesto da un'operazione per mezzo del numero di accessi a disco.

4. I record sono caratterizzati da un ristretto numero di attributi, meno di 10, ma il dominio di ognuno è ampio e linearmente ordinato.

Negli esempi che seguono, per rendere più semplice la visualizzazione, ipotizzeremo che la capacità di un cluster sia di tre record e che ognuno di essi sia costituito da due attributi.

[10]Con cluster si intende una unità di memoria di massa accessibile con una sola operazione di lettura/scrittura.

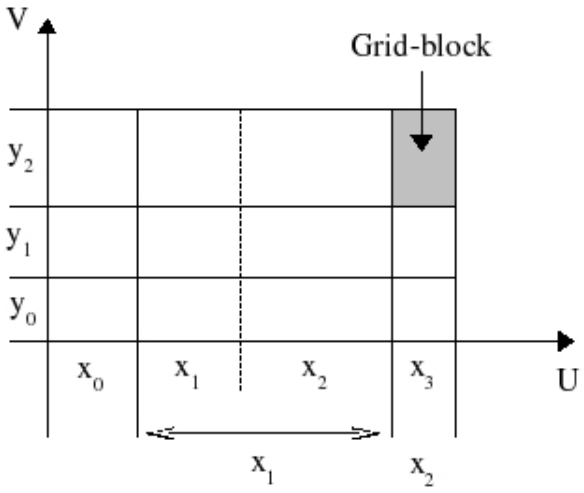

Figura 16: Esempio di partizione dello spazio dei record S.

6.1 Struttura dei Grid-File

Iniziamo definendo una partizione a griglia nel caso bidimensionale: sullo spazio dei record $S = U \times V$, si ottiene una *grid partition* $P = X \times Y$ per mezzo di due insiemi di intervalli organizzati in partizioni monodimensionali $X = \{x_0, x_1, \ldots, x_l\}$ e $Y = \{y_0, y_1, \ldots, y_m\}$, associate ognuna ad una dimensione, le quali dividono la spazio di ricerca in blocchi rettangolari chiamati *grid block*, come mostrato in Figura 16.

La griglia P è modificata alterando una delle sue componenti alla volta, ovvero, dividendo un intervallo di una partizione uno-dimensionale in due, oppure fondendo due intervalli adiacenti in uno; la Figura 16 ne mostra un esempio sulla dimensione X. Come si può notare, gli intervalli precedenti a quello diviso o ai due fusi insieme mantengono il loro indice, mentre l'indice degli intervalli successivi è aumentato di $+1$ o -1 rispettivamente.

Per ottenere una struttura di accesso ai dati basata su una partizione a griglia, occorre ora definire la corrispondenza dinamica tra grid-block e cluster, il cui compito è affidato alla *grid-directory*, che è una struttura dati costituita da:

- una matrice dinamica bidimensionale G di dimensione $n_x \times n_y$ con $n_x, n_y > 1$, denominata *grid-array*, i cui elementi sono puntatori a cluster e sono in corrispondenza uno a uno con i grid-block della partizione. Tale struttura è memorizzata nella memoria di massa.

- due array monodimensionali $\overline{X}$ e $\overline{Y}$ di dimensione rispettivamente n_x+1

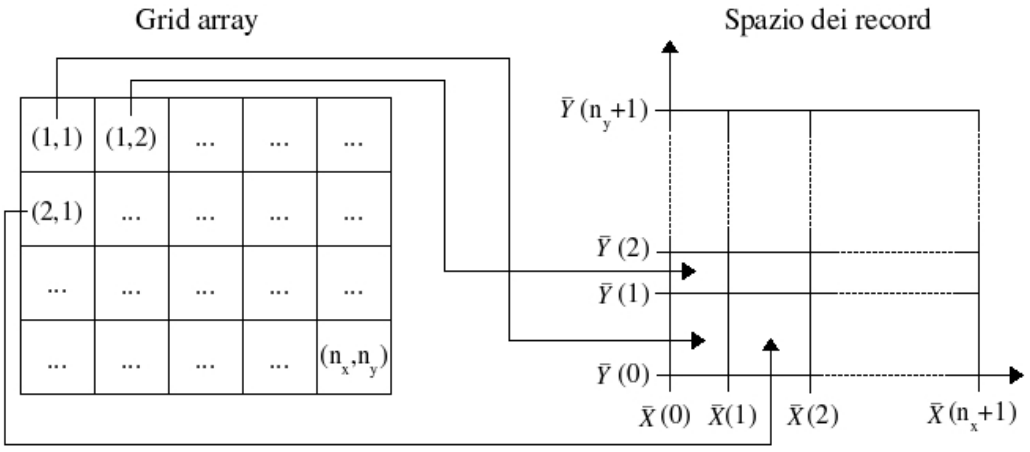

Figura 17: Corrispondenza tra *grid-array* e *grid-block*.

ed n_y+1 denominati *scale lineari*, di cui ognuna definisce una partizione del dominio S. Possono essere tenute in memoria centrale.

La Figura 17 mostra la corrispondenza tra grid-array e grid-block; da notare che i valori nelle scale lineari definiscono le rette che dividono lo spazio dei record.

Sulla *grid-directory* sono definite alcune operazioni:

- Accesso diretto: $G(c_x, c_y)$, con $0 < c_x \leq n_x$, $0 < c_y \leq n_y$.

- Funzione "Next" in ogni direzione :

  next$x$above: $c_x \leftarrow (c_x + 1)\, mod\, (n_x)$

  next$x$below: $c_x \leftarrow (c_x - 1)\, mod\, (n_x)$

  next$y$above: $c_y \leftarrow (c_y + 1)\, mod\, (n_y)$

  next$y$below: $c_y \leftarrow (c_y - 1)\, mod\, (n_y)$

- Merge

  merge$x$: dato p_x tale che $1 < p_x \leq n_x$, fusione di p_x con next$x$below; rinominare tutti gli elementi successivi a p_x; aggiornare la scala $\overline{X}$.

  merge$y$: simile a merge$x$ per p_y tale che $1 < p_y \leq n_y$.

- Split

split$x$: dato p_x tale che $0 < p_x \leq n_x$, creare un nuovo elemento $p_x + 1$ e rinominare tutte le celle successive a p_x; aggiornare la scala $\overline{X}$.

split$y$: simile a split$x$ per p_y tale che $0 < p_y \leq n_y$.

È permesso che due puntatori del grid-array facciano riferimento allo stesso cluster, cioè che ad un cluster siano associati più grid-block: a tal proposito, si definisce la regione di un cluster C come l'insieme di tutti i grid-block ad esso associati.

Dato che la forma delle regioni influisce su range-query ed aggiornamenti seguenti la modifica della griglia, si impone un vincolo: solo un sottoinsieme di assegnamenti tra grid-block e cluster sono ammessi, e in particolare si vuole che le regioni ad essi associate siano *convesse*. Tale condizione si esprime formalmente in termini di grid-array, dicendo che se $G(x^{'}, y^{'}) = G(x^{''}, y^{''})$ allora, per ogni x,y tali che $x^{'} \leq x \leq x^{''}$ e $y^{'} \leq y \leq y^{''}$, abbiamo $G(x^{'}, y^{'}) = G(x, y) = G(x^{''}, y^{''})$; dal punto di vista geometrico, una regione è convessa se per ogni coppia di punti al suo interno, tutti i punti sul segmento che li unisce appartengono alla regione data.

Alcune politiche di split e merge restringono ulteriormente l'insieme degli assegnamenti convessi.

6.2 Operazioni sui Grid-File

Prendiamo in esame, come per i k-d-Tree, tre tipi di ricerca, inserimento e cancellazione.

Point query, ricerca di un determinato record: vediamo a tal proposito un esempio che dovrebbe chiarire anche quanto detto finora.

Consideriamo uno spazio dei record con un attributo "anno" e un altro "mese". Supponiamo una distribuzione dei record tale da indurre una partizione a griglia del tipo:

$$\overline{X} = (0, 1000, 1500, 1750, 1875, 2000); \quad \overline{Y} = (0, 6, 7, 8, 12).$$

Si vuole cercare il punto $[1950, 11]$, se esiste: l'esecuzione della query è mostrata in Figura 18 (si assume che ci sia il record cercato). Il valore del primo attributo, 1950, è convertito nell' indice 5 per mezzo di una ricerca nella corrispondente scala lineare $\overline{X}$, dalla quale si scopre che 1950 è contenuto appunto nel quinto intervallo $[1875, 2000)$; similmente, 11 è convertito nell'indice 4 dopo una scansione della scala $\overline{Y}$. Questi due indici, presi nell'ordine, sono le coordinate grid-array del puntatore al cluster che eventualmente contiene il

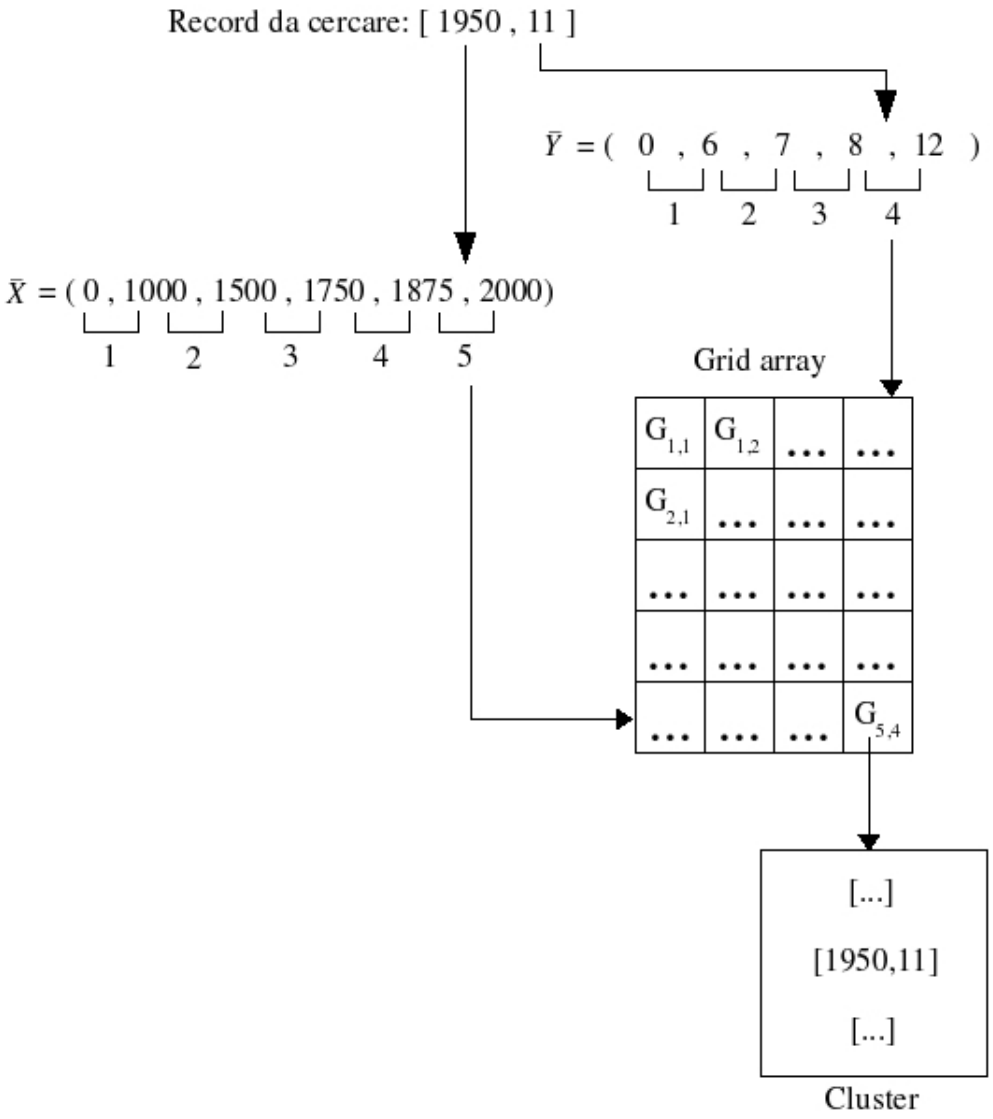

Figura 18: Esempio di ricerca del punto $[1950, 11]$.

punto cercato. Come convenzione si è deciso che gli intervalli di partizione siano semi-aperti a destra.

Un algoritmo ad alto livello per la ricerca puntuale consiste nei seguenti passi:

- per ogni attributo, ricerca sulla corrispondente scala lineare per recuperare l'indice dell'intervallo corretto. Siano x e y gli indici ottenuti dalla scansione delle scale $\overline{X}$ e $\overline{Y}$ rispettivamente.
- con le coordinate del passo precedente accedere alla cella $G(x, y)$ del grid-array che contiene l'indirizzo del cluster;
- accedere al cluster per recuperare eventualmente, attraverso una scansione sequenziale, il record cercato.

Il costo computazionale è di due accessi al disco di cui uno per il grid-array e l'altro per il cluster: si ricorda che non viene tenuto conto della scansione delle scale lineari poichè risiedono in memoria centrale.

Range query, ricerca dei record contenuti in una data regione R. Il principio è quello visto per i k-d-Tree e consiste nell'esaminare solamente i record dei grid-block che intersecano R.

Un algoritmo ad alto livello potrebbe essere questo:

- determinare, esaminando le scale lineari, quali grid-block intersecano R;
- se alcuni grid-block sono interamente contenuti in R, ritornare tutti i record in essi contenuti;
- per ogni grid-block interessato dall'intersezione verificare quali dei suoi record eventualmente stanno in R.

Nell'implementazione dell'algoritmo bisogna tenere presente che più di un grid-block può essere associato ad un singolo cluster e si deve usare un meccanismo che consenta di individuare quali di essi sono già caricati in memoria centrale per evitare di caricarli nuovamente, come potrebbe capitare nell'esempio di Figura 19. Un piccolo accorgimento è quello di introdurre un flag che renda conto dello stato di un cluster.

Nearest neighbour query, trovare il punto (x, y) più vicino al target. Seguendo lo stesso principio adottato per i k-d-Tree, la tecnica è quella di iterare una range-query con regione circolare R centrata nel target, di raggio via via decrescente in dipendenza del record più vicino trovato fino a quel momento; si parte con raggio teoricamente infinito, in pratica basta che R contenga la parte di piano in cui ci sono i record.

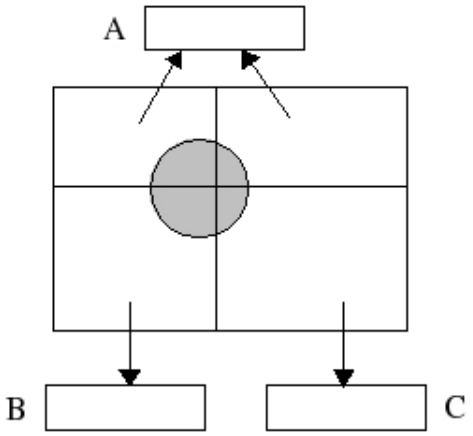

Figura 19: Il cluster A potrebbe essere considerato due volte.

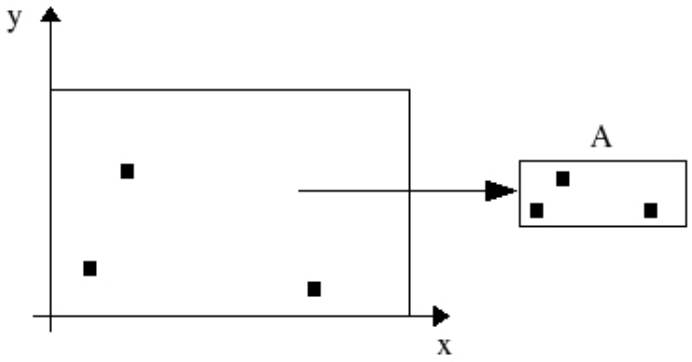

Figura 20: Inserimento dei primi tre punti.

Inserimento di un record: prima di introdurre l'algoritmo, vediamo attraverso un esempio le modificazioni della struttura che avvengono dopo l'inserimento di record. Si assume che la politica di split sia di dividere in modo alternato: una volta lungo un asse, la volta dopo lungo l'altro e così via.

Inizialmente, un singolo cluster A di capacità $c = 3$ è assegnato all'intero spazio dei record (Figura 20). Viene inserito un nuovo record p, il quale manda A in overflow: è necessario dividere lo spazio, allocare un nuovo cluster B e ridistribuire i valori nei due cluster. La Figura 21 mostra la configurazione che si ottiene dopo un ulteriore inserimento, quello di q. In seguito all'arrivo di r il cluster A va di nuovo in overflow e il grid-block a lui associato è diviso in accordo con la politica di split, con conseguente allocazione del cluster C e riassegnamento dei record come si vede dalla Figura 22. Da notare che siccome il cluster B *non* va in overflow, non subisce nessuna modifica e la regione a lui associata consiste ora di due grid-block;

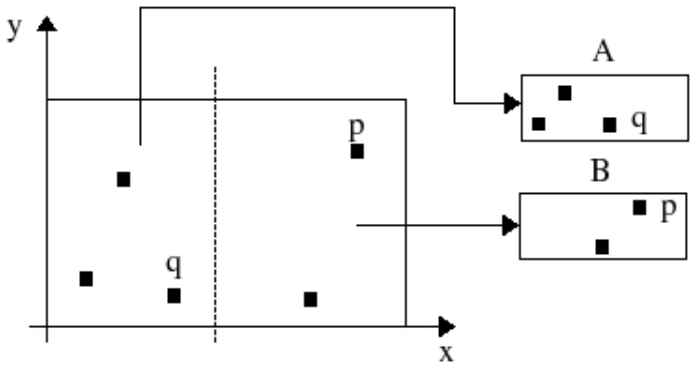

Figura 21: Inserimento dei punti p e q.

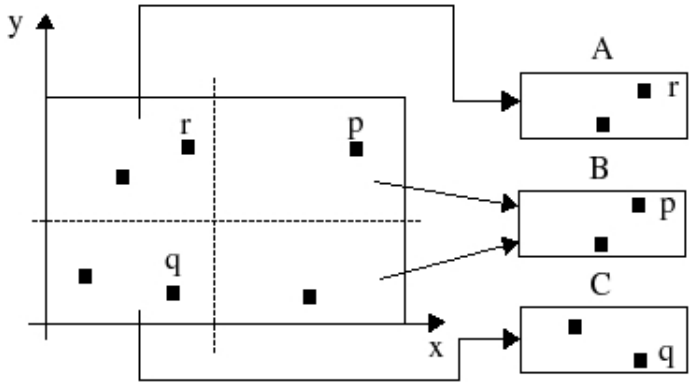

Figura 22: Inserimento di r.

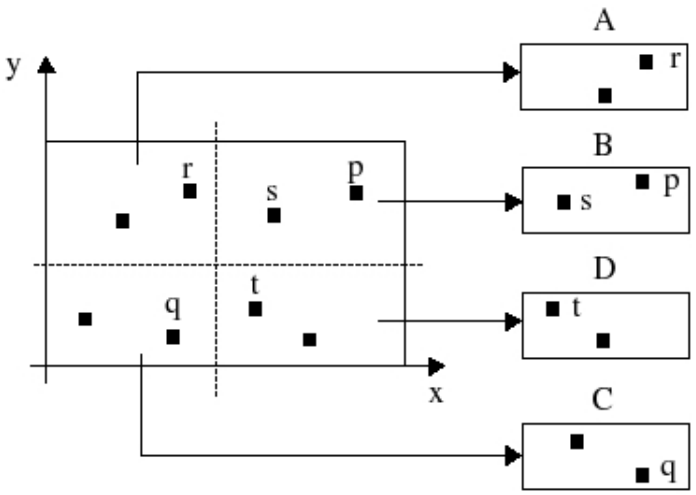

Figura 23: Inserimento di s e t.

si è visto che non è necessario creare un nuovo cluster quando la sua regione è divisa. Due nuovi record s e t sono inseriti; quest'ultimo provoca l'overflow di B; si alloca D e la configurazione finale è riportata in Figura 23.

Dall'esempio precedente sono emersi due tipi di comportamento che incorrono durante un inserimento: divisione e allocazione di un nuovo cluster oppure solo allocazione del cluster. Queste dinamiche prendono il nome di *divisione totale* e *divisione parziale* rispettivamente. Formalizziamo meglio i passi da seguire nei due casi:

- Divisione **totale**: si effettua quando la regione associata ad un cluster, che va in overflow, è costituita da *un* grid-block solamente.

  - Stabilire la dimensione su cui fare la divisione (dipende dalle politiche di split).
  - Determinare il nuovo valore m da aggiungere alla scala lineare corrispondente alla dimensione del punto precedente.
  - Modificare la scala lineare.
  - Creare un nuovo cluster.
  - Riordinare gli oggetti nei cluster coinvolti dall'operazione.
  - Modificare e aggiornare il grid-array.

  Per la seconda fase dell'algoritmo, si può ad esempio dividere a metà l'intervallo coinvolto oppure considerare il valore mediano dei relativi record in modo che ne cadano in egual numero da una parte e dall'altra della linea di split.

- Divisione **parziale**: si effettua quando la regione associata ad un cluster, che va in overflow, è costiuita da più di un grid-block.
    - Creare un nuovo cluster.
    - Riordinare gli oggetti nei cluster coinvolti dall'operazione.
    - Aggiornare il grid-array.

L'algoritmo ad alto livello per l'inserimento consiste nei seguenti passi:

- ricerca del grid-block in cui il record dovrebbe andare a finire;
- verifica se il cluster associato va in overflow;
- se non c'è overflow inserire il record, altrimenti effettuare divisione totale o parziale a seconda che l'overflow del cluster coinvolga uno o più grid-block rispettivamente.

Cancellazione di un record:

- ricerca del record da cancellare;
- verifica se il cluster che lo contiene va in underflow;
- se non c'è underflow eliminare il record e fine, altrimenti effettuare il merge con un bucket adiacente, modificare eventualmente[11] ed aggiornare la grid-directory.

Le politiche di merge coinvolgono alcune decisioni, come ad esempio individuare quale coppia di cluster adiacenti sono candidati per essere fusi a formare un singolo cluster; se esistono più coppie candidate, quale ha la priorità; sotto quale soglia effettuare il merge.

Riferimenti bibliografici

[1] P. Atzeni, S. Ceri, P. Fraternali, S. Paraboschi, R. Torlone. *Basi di dati Architetture e linee di evoluzione*. McGraw-Hill, 2003, ISBN 88-386-6030-1.

[2] Ramez A. Elmasri, Shankrant B. Navathe. *Fundamentals of Database Systems*, third edition. Addison-Wesley, 1999, ISBN 08-053-1755-4.

[11] Non è detto che l'underflow di un cluster comporti la modifica delle partizioni.

[3] Andrew Moore. *An introductory tutorial on kd-trees*, extract from A. Moore's PhD Thesis: *Efficient Memory-based Learning for Robot Control*. Computer Laboratory, University of Cambridge, 1991.

[4] J. Nievergelt, H. Hinterberger, K. C. Sevcik. *The Grid File: An Adaptable, Symmetric Multikey File Structure*. ACM Transactions on Database Systems, Vol.9, No.1, Marzo 1984, Pagine 38-71.

[5] Tesi di Laurea in Ingegneria Informatica di Manuele Piastra: *"CURE+": un'implementazione dell'algoritmo di clustering gerarchico agglomerativo "CURE" con riferimento all'analisi di grandi volumi di dati*. Università di Pisa, 18 Dic. 2003.

[6] Appunti del corso di *Basi di dati e multimedia* tenuto dai docenti prof. Alberto Belussi e prof. Carlo Combi, 2006.

[7] *Appunti del corso di informatica medica* di Donatella Gubiani. Università di Udine, Anno Accademico 2000-2001.

[8] *Lecture 08: Multi-dimensional Indexing*. ICS 214A: Database Management Systems Winter 2004.

`www.ics.uci.edu/~ics214a/handouts/slides08.pdf`.

[9] Presentazione: *Indexing Low and High Dimensional Spaces*.

`www.cs.ust.hk/~leichen/courses/comp630j/lecturenotes/comp630-lecture2.pdf`