Tecnologie delle Basi di Dati M

Antonio Davide Calì

10 maggio 2014

WWW.ANTONIOCALI.COM
Anno Accademico 2013/2014
Docenti: Marco Patella, Paolo Ciaccia

Indice

Parte I
Strutture di Indicizzazione n-Dimensionali

Finora abbiamo parlato di strutture tradizionali, con richieste che tipicamente si trovano tutti i giorni su database relazioniali. Affrontiamo ora un argomento a margine, ovvero le struttre di indicizzazione n-dimensionali.

Come abbiamo visto, B+tree è in grado di risolvere interrogazioni che coinvolgono più attributi: infatti, B+tree per indici multi-attributo sfrutta un'ordinamento lessicografico degli attributi e di conseguenza riesce a risolvere le query

(in realtà riesce a risolvere non solo le query su tutti gli attributi, ma anche solo su un prefisso della chiave). L'efficienza in risoluzione è suffiginte?

Esistono diversi tipi di interrogazioni n-dimensionali:

1. **point query**: dati da sole uguaglianze $A_1 = v_1$, $A_2 = v_2$, ..., $A_n = v_n$ in cui ogni attributo ha un valore esatto. Se pensiamo a una tupla in uno spazio a n-dimensioni, una *point query* altro non è che trovare il punto delle cordinate $(v_1, v_2, \ldots, v_n)$ e quindi come una ricerca puntuale.

2. **window query**: in cui sono presenti intervalli $l_1 \leq A_1 \leq h_1$, $l_2 \leq A_2 \leq h_2$, ..., $l_n \leq A_n \leq h_n$, di conseguenza una ricerca *window query* è chiedere le tuple presenti all'interno di una "finestra" (dove per finiestra intendiamo l'iper-rettangolo che viene definito dall'interesezione di tutti gli intervalli). **NB** Una *point query* è un caso particolare di *window query* in cui i due estremi dell'intervallo coincidono.

3. **nearest neighbor query**: $A_1 \approx v_1$, $A_2 \approx v_2$, ..., $A_n \approx v_n$. È una ricerca particolare che, sapendo che una tupla con esettamente i valori $(v_1, v_2, \ldots, v_n)$ non esiste, richiede di trovare una (o più) tupla il più possibile vicina a quella "target". In una situazione del genere è richiesto anche di implementare una definizione di "distanza" fra tuple per riuscire ad indicare la tupla più vicina.

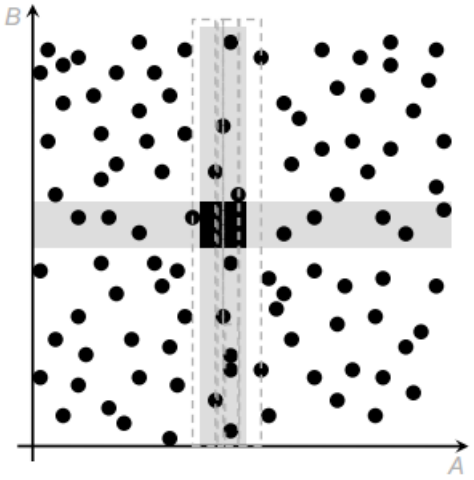
E se il valore del dato non fosse "puntuale"? Con ciò intendiamo,ad esempio, che una tupla possa avere più valori sugli attributi. Di conseguenza la tupla non viene rappresentata da un unico punto, ma può essere vista, ad esempio in due dimensioni, come un segmento. Le cose dati "non puntuali" si complicano, in quanto anche la ricerca window query non può essere più effettuata controllando singoli punti. Si potrebbe pensare di poter controllare solo gli estremi del segmento, ma questo non risulta sufficiente poichè gli estremi potrebbero essere al di fuori della "finestra" richiesta dalla *window query* ma l'intersezione fra il segmento e la finestra stessa potrebbe non essere ugualmente vuota.

Alcuni esempi di utilizzo di strutture di indicizzazione n-dimensionali possono essere *sistemi informativi* geografici/spaziali, quindi con coordinate di punti per identificare luoghi e città o con oggetti con estensione, come regioni, vie fiumi, oppure possono essere utili per *basi di dati multimediale* in cui viene effettuata una ricerca basata sul "contentuto" in cui è necessaria la rappresentazione del contenuto per mezzo di caratteristiche numeriche e la similarità del contenuto si ottiene valutando la similarità delle caratteristiche (cioè se due oggetti hanno caratteristiche simili allora si ritiene che abbiano anche contenuti simili).

Vediamo come sia possibile utilizzare il B+tree. Supponiamo di avere una *window query* su due attributi (A,B) in cui ogni intervallo rappresenta il 10% del totale (cioè entrambi gli intervalli A e B della query abbiano una selettività del 10%) e in cui ci aspettiamo di recuperare l'1% dei dati (dato dal prodotto delle selettività dei singoli interevalli). Le possibili soluzioni sono:

1. L'uso di 1 singolo B+tree *bi-dimensionale* (A,B) e si controlli, in figura (Fig. 1), come i dati distribuiti vengono recuperati (si suppone che la capacità delle foglie sia di 3 record).

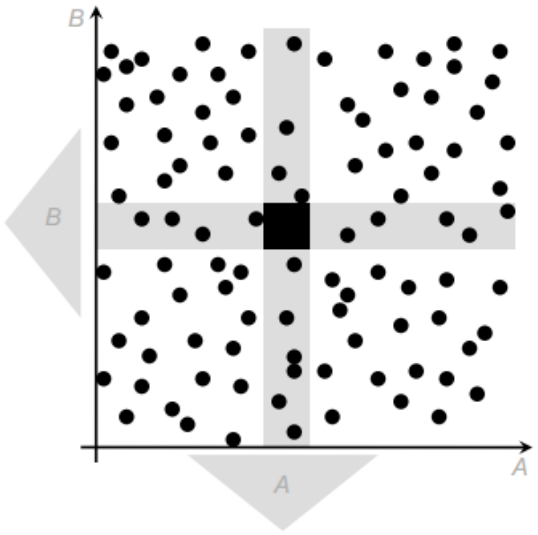
Figura 1:



Si vede che si accede prima all'intervallo di A, quindi al 10% delle foglie, attraverso l'indice, ma non il fatto che l'intervallo di B abbia anche esso una selettività del 10% è del tutto superfluo, poichè accedo ugualmente a tutte le foglie che A mi restituisce e solo sui dati che recupero da queste foglie riesco a controllare la selezione su B. Di conseguenza avere anche la selezione su B non riesce a diminuire in alcun modo il numero di foglie a cui si vuole accedere.

2. L'uso di 2 B+tree *mono-dimensionali* (A), (B) e si controlli, in figura (Fig. 2), come i dati vengono recuperati. In questo caso si accede al 20% dei dati.

Figura 2:



In questo caso il risultato è peggiore. Infatti accedere prima al 10% dei dati dovuti alla lettura di A e poi ad altri 10% dei dati dovuti alla letture delle foglie grazie a B e solo successivamente effettuare la vera selezione, mi richiede di aver letto il 20% dei dati quando in realtà la selettività che ci eravamo immaginati era del solo 1%.

Si noti che aumentando la dimensione da due a 3, ci si aspetta che la selezione della window query sia dello 0,1% e utilizzando un B+tree tri-dimensionale si avrebbe ancora il 10% di letture di foglie, mentre utilizzando tre B+tree mono-dimensionali richiederebbe il 30%. Come si vede all'aumentare della dimensione le performance tendono a peggiorare notevolmente (pensando soprattutto alla selettività che ci si aspetta dalla query).

In entrambi i casi troppo lavoro viene sprecato. La causa è nel fatto che punti vicini nello spazio sono assegnati a foglie lontane, nel primo caso dalla "linearizzazione" degli attributi (dovuto all'ordinamento lessicografico), cioè al tentativo di dare un ordine totale a uno spazio a due dimensioni, nel secondo dall'ignorare completamente il secondo attributo. Gli indici multi-dimensionali (*spaziali*) cercano di mantenere la prossimità spaziale dei record.

1 Indicizzazione spaziale

Il problema nasce negli anni '70 a causa dell'insorgere di problemi 2/3-D come la cartografica, il GIS (Geographic Information System), il VLSI, il CAD, ... Venne recuperato successivamente negli anni '90 per risolvere i problemi posti da nuove applicazioni come i Database multimediali e il Data mining.

Esistono diversi approcci per gli indici spaziali:

1. Derivati da struttere 1-D come ad esempio k-d-B-Tree, EXCELL, Grid file
2. Mapping da n-D a 1-D come ad esempio Z-order, Gray-order. Si noti, come già detto, che si cerca di dare un ordine totale (cosa possibile in 1-D) a strutture che ordine totale non hanno (appena si supera la prima dimensione). Evidentemente non si riesce a mentenere il principio di *località* che discuteremo a breve.
3. Struttre create ad-hoc per gestire n dimensioni come ad esempio R-tree, R*-tree, X-tree, ... Si noti che di strutture di questo tipo ne esistono a centinaia.

Classificazione Possiamo classificare gli indici spaziali per il tipo di oggetti, tipo di suddivisione o tipo di organizzazione.

- **Tipo di oggetti**
  - Per *punti* (i record non possono avere un'estensione spaziale)
  - Per *regioni*. I punti non sono altro che un caso specifico di regioni. Come abbiamo già detto precedentemente questo può accadere per tuple multivalore, o ad esempio se si vuole indicizzare un oggetto con estensione come ad esempio un lago, un fiume.
- **Tipo di suddivisione**
  - Sullo *spazio* (le divisioni avvengono sulla base di considerazioni globali, stile linear hashing, in cui ad esempio facevamo lo split di un bucket indipendentemente dal bucket che andava in overflow). Si usano per distribuzioni uniformi, semplice da gestire
  - Sugli *oggetti* (le divisioni avvengono sulla base di considerazioni locali, stile B-tree, cioè si divide lo spazio dove è più necessario). Si usano per distribuzioni arbitrarie, complicate da gestire.
- **Tipo di organizzazione**
  - Basata su *albero* (stile B+tree)
  - Basata su *hash*

Ci sono alcune generalità sugli indici spaziali che valgono sempre. Un *requisito di base* è il *Local Order Preservation* (cioè di preservare un ordine locale dei dati). Per far ciò si deve:

- Raggruppare gli oggetti (punti) in pagine, garantendo che all'interno di ciascuna pagina vi siano oggetti "vicini" nello spazio n-D

- Escludere l'uso di funzioni hash che non preservano l'ordine (sembra contraddittorio con quanto detto sul tipo di organizzazione, ma effettivamente si utilizzeranno principi simili ad una funzione hash, senza effettivamente usarla).

Il problema è non banale in quanto in n-D non è definito un ordine globale e dunque alcune soluzioni definiscono un ordine in n-D per riuscire a risolvere il problema (almeno in parte).

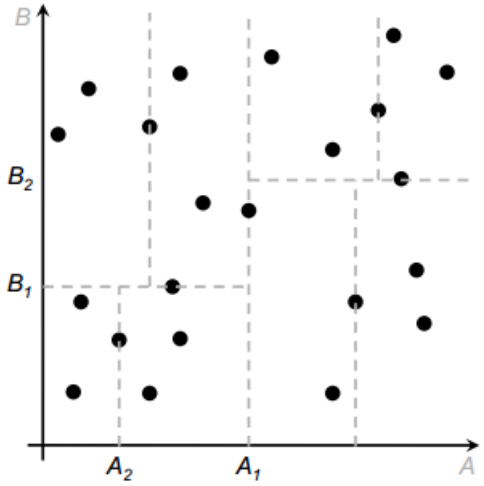
Un approccio generale agli indici spaziali è di organizzare lo spazio in *regioni* (o celle) in cui ogni *cella* corrisponde (non sempre con corrispondenza uno a uno) con una pagina.

1.1 k-d-tree

Il k-d-tree (Bentley, 1975) è una struttura pensata per la memoria centrale, dunque non è paginata e non è bilanciata. È sostanzialmente un albero binario di ricerca in cui ogni livello è *etichettato* (ciclicamente) con una delle n coordinate e in ogni nodo c'è un *separatore* dato dal valore mediano dell'intervallo che si sta partizionando. Partendo dalla radice, che ad esempio etichettiamo con A1 (quindi andrà a dividere i dati per l'attributo A), che risulta essere il valore mediano dei valori di A, compariranno nel suo figlio sinistro le tuple tali che $A \leq A1$ e nel figlio destro le tuple tali che $A > A1$. Al livello successivo, essendo partiti con la radice etichettata per A, dobbiamo ora etichettare i nodi per B: indicato con B1 il valore mediano dei valori di B allora nel figlio sinistro compariranno le tuple tali che $B \leq B1$ e nel figlio destro le tuple tali che $B > B1$ e così procedendo fino ad avere una partizione dei dati in modo che ogni foglia riesca a contenere (fisicamente) i dati.

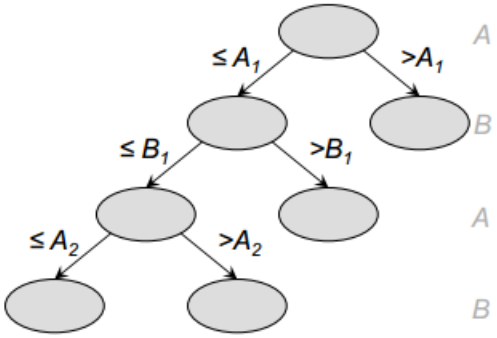
Supponiamo che ogni foglia possa contenere fino a 3 oggetti che sono disposti nello "spazio" come nella seguente figura (Fig. 3)

Figura 3:



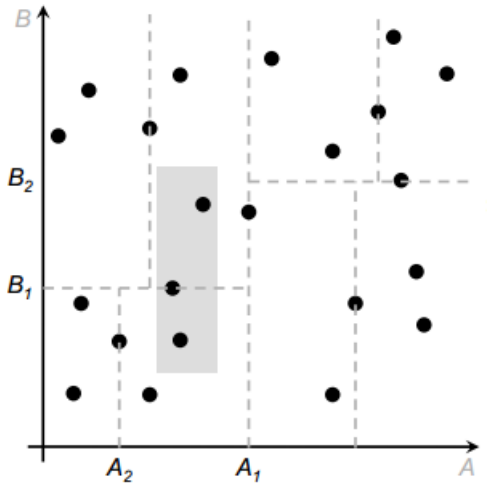
Il *k-d-tree* corrispondente che si crea è mostrato in figura (Fig. 4), in cui A1 è il valore mediano di tutti i valori di A, B1 è il valore mediano di tutti i valori di B, A2 è il valore mediano dei valori di A che stanno nella porzione di piano a sinistra di A1, ...

Figura 4:



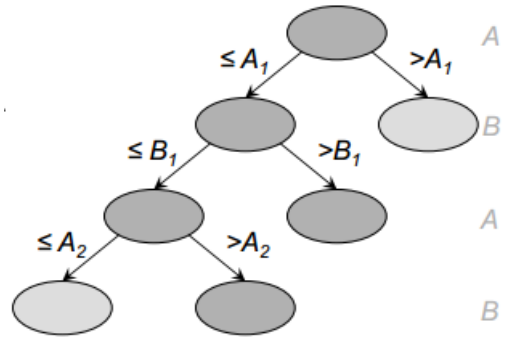
Per effettuare una ricerca si seguono tutti i rami interessati dalla query, ad esempio se si vogliono cercare i 3 oggetti mostrati in figura (Fig. 5)

Figura 5:



allora la ricerca viene effettuata sull'albero come segue (Fig. 6). La ricerca deve controllare tutte le foglie che hanno un'intersezione non vuota con la window query.

Figura 6:



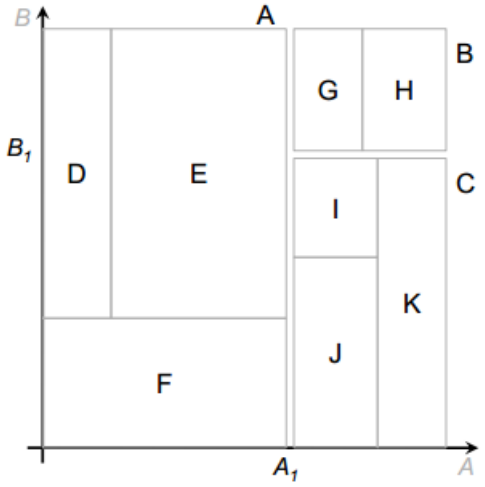
Quando si deve inserire un nuovo elemento si cerca la foglia che deve contenerlo e se la foglia è piena (overflow) avviene uno *split* verso il basso (come accade in un albero binario). L'albero non è bilanciato, come si vede, e periodicamente richiede una riorganizzazione. Le cancellazioni sono estremamente complicate, soprattuto se si vogliono effettuare dei *merge* delle foglie sotto utilizzate (underflow). Esistono diverse varianti per gestire in maniera diversa i separatori come ad esempio il *BSP tree* che utilizza iperpiani non paralleli agli assi oppure il *VAMsplit kd-tree* che scelgono la coordinata migliore di split (e quindi non il valore mediano) per ogni nodo come quella avente la maggior varianza.

1.2 k-d-B-tree

Il *k-d-B-tree* (Robinson, 1981) è una versione paginata del *k-d-tree*. La struttura risultante assomiglia molto ad un B+tree. Ogni *nodo* (che corrisponde ad una pagina) corrisponde ad una *regione* (iper-) rettangolare dello spazio, chiamata *box* o *brick*, ottenuta come unione delle regioni figlie avente assi paralleli agli assi cartesiani. Se nel B+tree in ogni nodo interno avevo più separatori (numerici) che mi permettevano di distinguere i figli, nel k-d-B-tree avviene praticamente la stessa cosa, solo che invece che separatori numerici utilizziamo dei iper-rettangoli. Internamente i nodi sono organizzati come dei *k-d-tree*: si noti che la "grandezza" dell'albero dipende dalla capacità di una pagina.

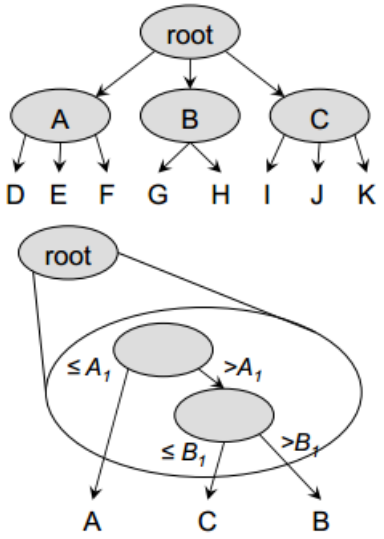
Mettiamo di aver suddiviso i dati come segue attraverso i vari box (Fig. 7)

Figura 7:



Il *k-d-B-tree* che ne deriva è mostrato in figura (Fig. 8), in cui si vede la divisione ad albero bilanciato e la rappresentazione di un singolo nodo (in questo caso la radice) come un *k-d-tree*.

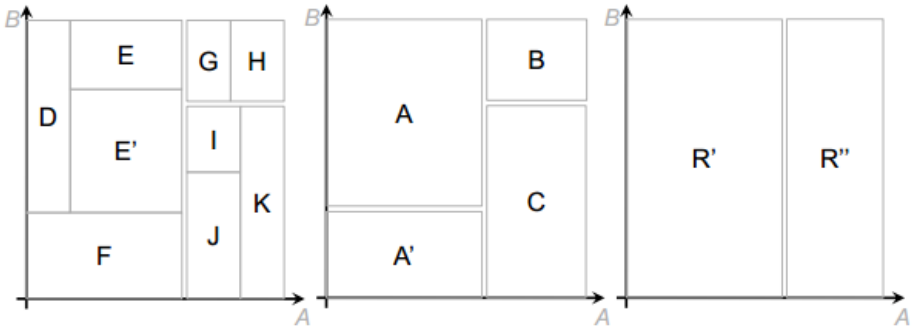
Figura 8:



Da come si può vedere dall'immagine di esempio la root è strutturata come un k-d-tree, quindi sarà possibile che al suo interno ci siano dei "nodi interni", come in questo caso avviene, per dividere e indirizzare i suoi figli.

Se un nodo indice (regione) va in *overflow* la situazione risulta più complessa che nel B-tree poichè sarà necessario sia uno split verso il basso (poichè è un k-d-tree) sia uno split verso l'alto (poichè è un B+tree). Si veda il seguente esempio (Fig. 9) in cui è richiesto lo *split* del blocco dati E che richiede la partizione di E, quindi di A e infine anche della radice.

Figura 9:



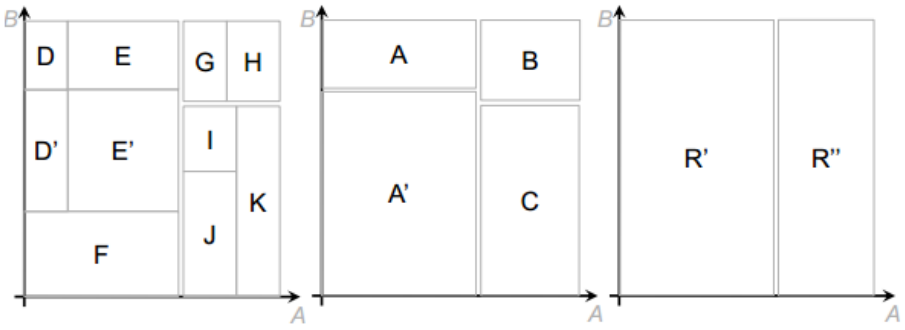
Quando avviene uno *split* la ridistribuzione che ne segue non è sempre bilanciata, poichè non è sempre possibile. Non vi è dunque nessun limite inferiore sull'utilizzazione di memoria allocata (che si aggira intorno al $\sim 50 - 70\%$).

Come si vede dall'esempio precedente, A è stata partizionata in A e A' sulla base del *primo* separatore.

Esiste un algoritmo migliore per effettuare la separazione, detto Algoritmo di *Robinson* il quale considera un iperpiano che distribuisca in modo *equilibrato* i nodi, ma che richiede che lo split si propaghi non solo verso l'alto (come avviene notoriamente nel B+tree) ma anche verso il basso, verso i discendenti.

Rivediamo l'esempio (Fig. 10) in cui questa volta A viene in A e A', ma anche D (figlio di A) viene suddiviso in D e D'.

Figura 10:



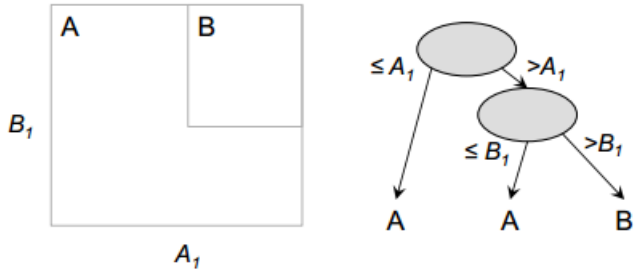
1.3 hB-tree

L'*hB-tree* (Lomet & Salzberg, 1990) è un ulteriore variante del *k-d-B-tree* in cui le regioni possono ora contenere dei "buchi" (hB sta per "*holey brick*"). Questo comporta alcuni effetti positivi come:

- *Split di un blocco dati*: Si riesce a garantire che i dati si ripartiscano al peggio in un rapporto 2:1 (ad esempio $\frac{2}{3}$ in un blocco e $\frac{1}{3}$ nell'altro).
- *Split di un nodo indice*: Si ottiene un bilanciamento (e quindi un limite inferiore all'occupazione di memoria) senza dover propagare gli split ai discendenti.

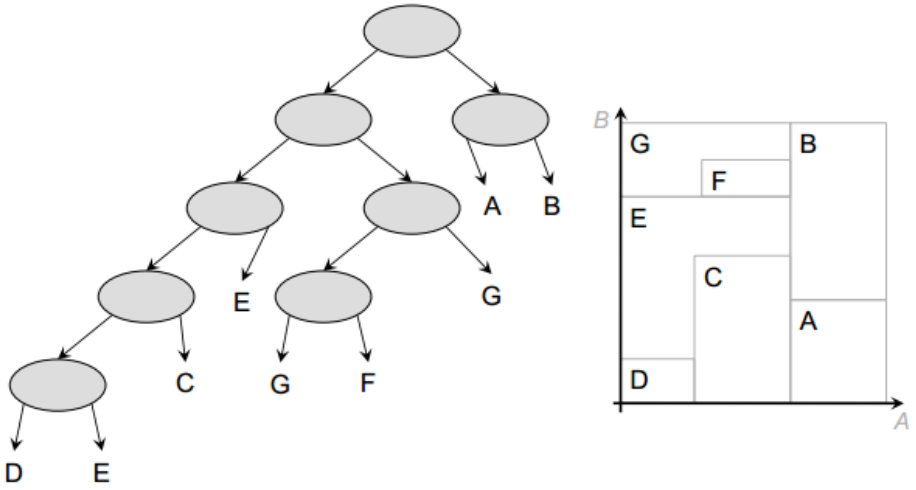
Come nel *k-d-B-tree* ogni nodo è internamente organizzato come un *k-d-tree* con la differenza che ogni nodo può essere "puntato" da diverse separazioni. Si guardi l'esempio (Fig. 11) in cui viene mostrato sia lo spazio dei dati sia l'*hB-tree* che ne deriva, in cui si vede il nodo A essere puntato da due nodi indice.

Figura 11:



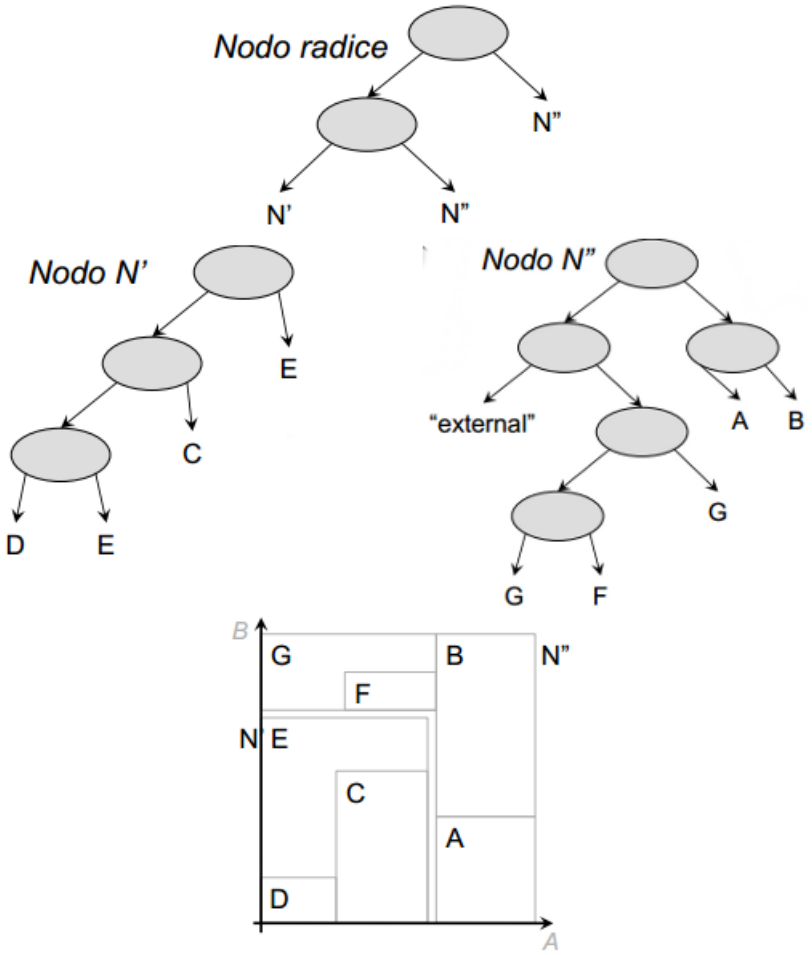
Vediamo come funziona lo split. Supponiamo che ogni pagina possa contenere al più 7 nodi e che la radice vada in *overflow*. La situazione iniziale è la seguente (Fig. 12)

Figura 12:



La situazione successiva allo split della radice diventa la seguente (Fig. 13)

Figura 13:

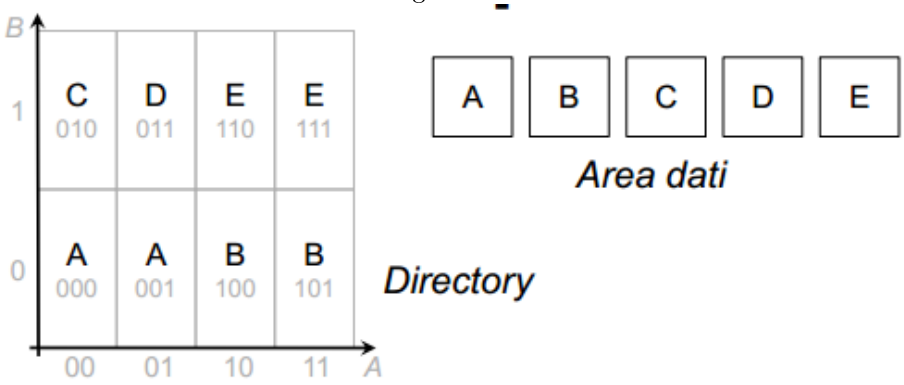


1.4 EXCELL

EXCELL (Tamminen, 1982), estensione su n-dimensioni dell'*extandable hashing*, usa una directory *hash-based* (cioè ha lo stesso comportamento di una funzione hash, ma non la utilizza), ovvero suddivide lo spazio in una griglia regolare in n dimensioni. Ogni *cella* della directory corrisponde a una pagina dati, ma non necessariamente vicecersa (come accade nell'extandable hashing). L'indirizzo di una cella è formato facendo l'*interleaving* dei bit delle coordinate, cioè alternando ciclicamente un bit di ogni coordinata. All'interno di ogni cella c'è un puntatore a un bucket, solo che in questo caso le stringhe di bit di ogni cella non sono ottenute con un generatore pseudo-causale (come accadeva nell'extendable hashing) ma ottenute dividendo lo spazio in maniera regolare: proprio per questo motivo posso avere duplicazione di puntatori.

Vediamo un esempio (Fig. 14) in cui l'area dati viene "mappata" nella directory

Figura 14:



Si vede dall'immagine che ogni indirizzo, ad esempio l'indirizzo di C, viene ottenuto alternato i bit, infatti si ottiene col primo bit della coordinata A (0), il primo bit della coordinata B (1) e infine il secondo bit della coordinata A (0) ottenento 010.

Quando una pagina dati va in *overflow* si esegue uno split e si distinguono due casi per la directory:

1. Se il blocco era referenciato da due (o più) celle si modificano solo i puntatori
2. Altrimenti si raddoppia la directory utilizzando un bit in più

Vediamo un esempio. Si prenda l'area dati e la directory mostrate nella figura precedente (Fig. 14) e si supponga che la cella A vada in overflow. Dato che è referenziata da più celle, semplicemente A viene divisa in A e F ed è sufficiente modificare il puntatore della cella 001 facendola puntare a F. La situazione che si crea è quella mostrata in figura (Fig. 15)

Immaginiamo successivamente che C vada in *overflow*, ne è richiesto lo split ma in questo caso C è referenziato da una sola cella, dunque viene divisa in C e G ed occorre raddoppiare la directory usando un bit in più per la coordinata B. La situazione finale è la seguente (Fig. 16)

Figura 15:

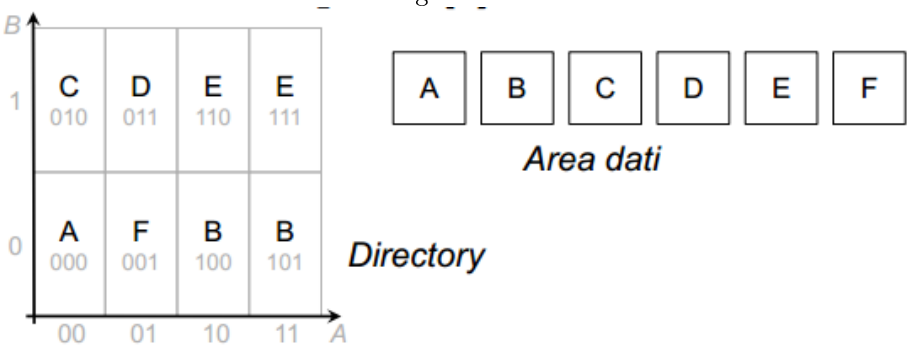
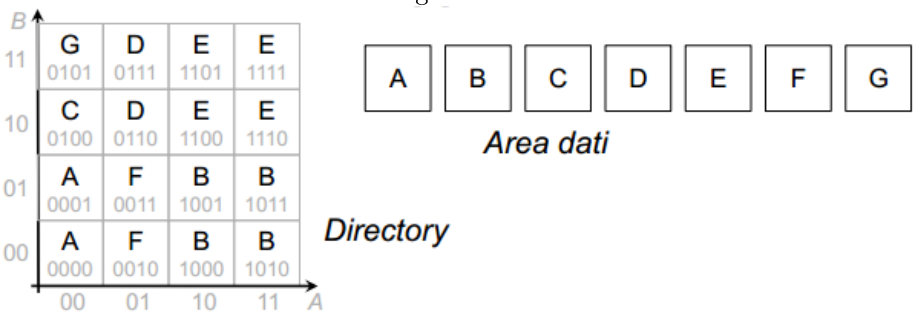


Figura 16:



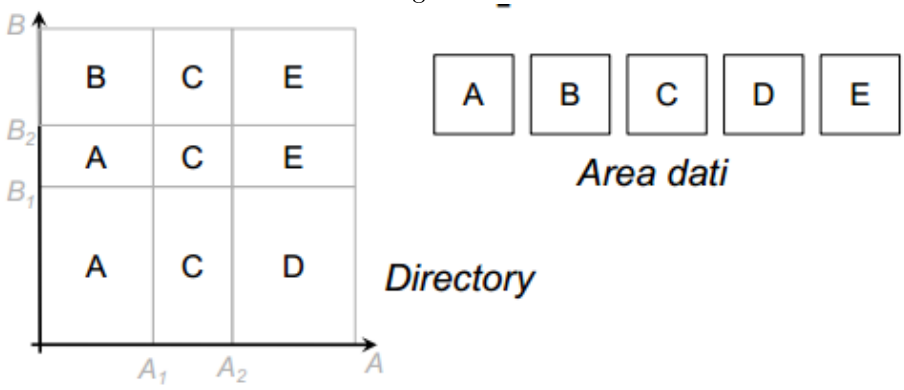
Per EXCELL valgono gli stessi ragionamenti visti per *extendible hashing*: si noti però che il raddoppio della directory non sempre riesce a risolvere l'*overflow* di un bucket, per lo stesso motivo per cui poteva non risolverlo nell'extendible hashing. EXCELL funziona molto bene in caso di distribuzione uniforme dei dati.

1.5 Grid file

Il *Grid File* (Nievergelt et al., 1984) generalizza *EXCELL*, permettendo la definizione di intervalli di ampiezza arbitraria. A tale scopo sono necessarie *d scale* che contengono i valori usati come separatori in ciascuna dimensione, non usando infatti più i bit per dividere lo spazio in maniera uniforme, ma valori come separatori ho bisogno di mantenere informazioni su questi separatori. Nel caso di intervalli definiti mediante partizionamente binario, le scale sono di fatto analoghe alle directory del dynamic hashing.

Un esempio di *Grid File* è mostrato in figura (Fig. 17). Si vede fin da subito che ora lo directory non è più suddivisa in un numero di celle potenza di due, proprio perchè non siamo più collegati ai bit.

Figura 17:

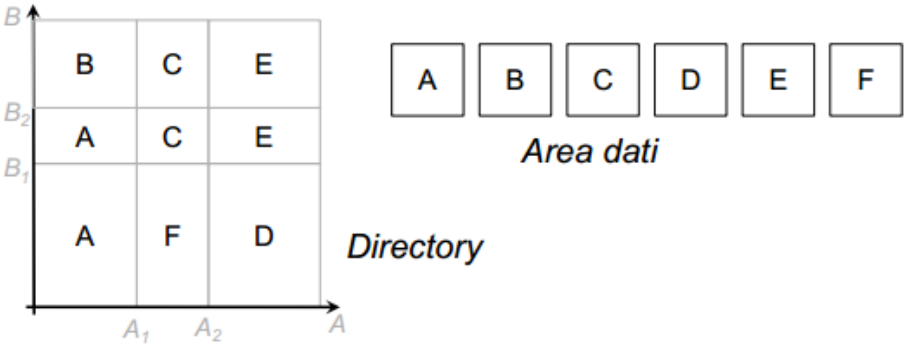


Quando una pagina dati va in *overflow*, si esegue uno split e si distinguono due casi per la directory:

1. Se il blocco era referenziato da due (o più) celle si modificano solo i puntatori
2. Altrimenti si aggiunge un separatore alla directory.

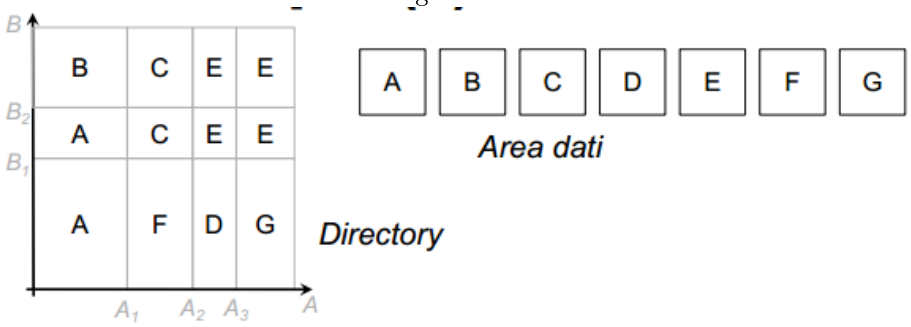
Si prenda da esempio la figura prima illustrata (Fig. 17) nel quale avviene un *overflow* da parte di C: la cella C viene divisa in C e F ed è sufficiente modificare il puntatore della cella a F. Il risultato finale è mostrato in figura (Fig. 18)

Figura 18:



Supponiamo che successivamente D vada in *overflow*: poichè esiste solo una cella che la referenzia è richiesto di suddivedere la directory. Dunque D viene divisa in D e G e occorre aumentare la directory suddividendo ulteriormente, ad esempio, la coordinata A. Il risultato finale è mostrato in figura (Fig. 19)

Figura 19:



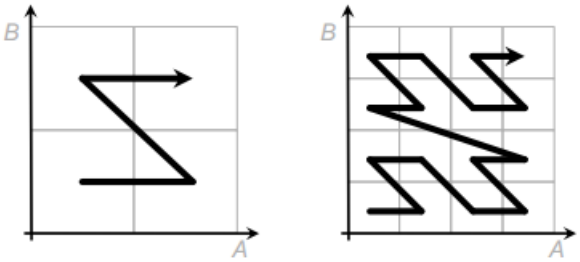
In caso di distribuzioni non uniformi, la gestione di N punti può richiedere un numero di celle proporzionale a $O(N^d)$, questo può accadere se i valori ad esempio sono distribuiti su una diagonale, richiederebbe appunto la suddivisione in N^d celle. La struttura regolare del partizionamente dello spazio semplifica però la risuolzuione di *window query*. Si nota anche che quando si introduce un nuovo valore separatore bisogna tener conto che si hanno effetti non solo "locali" sul bucket di overflow, ma anche su tutti i bucket della stessa colonna o stessa riga (a seconda del separatore che sto introducendo). Il problema principale rimane l'organizzazione della directory: normalmente le scale sono mantenute in memoria centrale. In casi (quasi-)statici la directory può essere memorizzata come array multi-dimensionale su disco, invece in casi dinamici è necessario paginare la directory, il che porta ai *Grid File multi-livello.*

1.6 Ordinamento mono-dimensionale

Si potrebbe pensare di utilizzare strutture monodimensionali per strutturare lo spazio. Si cerca dunque di "linearizzare" lo spazio n-dimensionale in modo da poter utilizzare struttre monodimensionali come B+tree. Si ottengono delle curve che "riempono lo spazio" chiamate *space-filling curve*: perchè esse siano utili devono fare in modo che punti vicini nello spazio siano vicini anche nella linearizzazione (requisito di *località*).

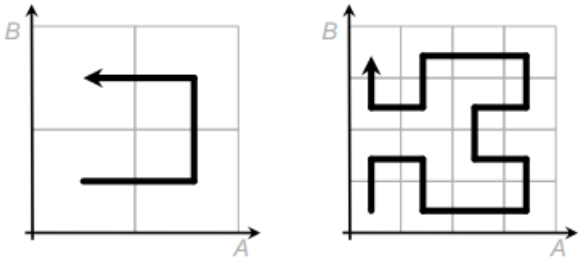
Esistono diversi tipi di *space-filling curve*, ne mostriamo di seguito alcuni esempi

Figura 20:



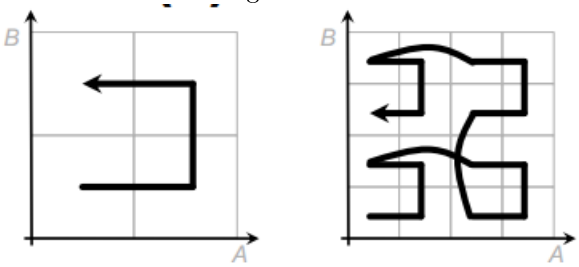
Z-order

Figura 21:



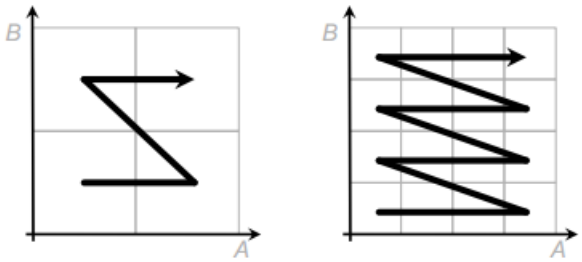
Peano-Hilber

Figura 22:



Gray-order

Figura 23:



Ordine lessicografico

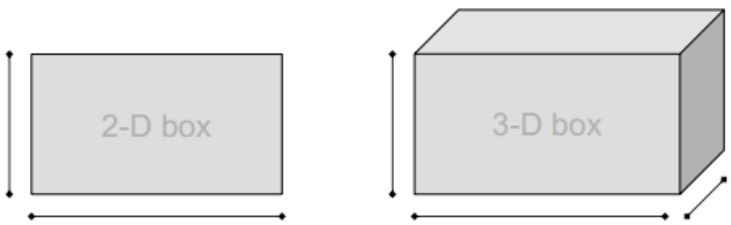
Come è evidente il requisito di località non è soddisfatto da nessuna delle curve. Pertanto la ricerca per *window query* soffre molto degli stessi problemi del B+tree multi-attributo: si possono notare infatti analogie. Inoltre la ricerca *nearset neighbor* si complica ulteriormente.

1.7 R-tree

L'*R-tree* è una struttura ad albero *bilanciata* e *paginata*, basata sull'innestamento gerarchico (cioè il padre è costruito a partire dai figli) di *overlapping regions*. Ogni nodo corrisponde ad una *regione rettangolare*, definita come l'*MMB* che contiene tutte le regioni figlie. L'utilizzazione di ogni nodo varia tra il 100% e un valore minimo ($\leq 50\%$) che è un parametro di progetto dell'R-tree. Utilizza meccanismi di gestione simili a quelli del B+tree con la differenza che l'inserimento di un oggetto e gli eventuali split che ne conseguono possono essere gestiti con politiche diverse. È una struttura estremamente simile a un B+tree, tanto che il B+tree ne è un suo caso particolare, quando si ricade in una dimensione e l'MMB altro non diventa che l'intervallo utilizzato nei B+tree. Il fatto che ci siano *overlapping regions*, cioè regioni che si sovrappongono, è dovuto al fatto che stiamo trattando uno spazio n-dimensionale in cui non è presente un ordinamento totale degli elementi.

L'*MMB* (*Minimum Bounding Box*) è il più piccolo rettangolo (Minuum), con i lati paralleli agli assi coordinati (Box), che contiene tutte le regioni figlie (Bounding). È definito come un prodotto di n intervalli (si veda la figura 24).

Figura 24:



Un'(iper-)rettangolo (che non è un MMB, poichè un iper-rettangolo non ha gli assi obbligatoriamente gli assi paralleli agli assi cartesiani) n-dimensionale ha 2^n vertici: per poter definire un (iper-)rettangolo basta specificare le coordinate di tutti i suoi vertici. L'algoritmo per il calcolo dell'(iper-)rettangolo più piccolo che contenga un insieme di N punti ha complessità $O(N^2)$ in 2-dim, $O(N^3)$ in 3-dim, ma non è noto alcun algoritmo per dimensioni maggiori di 3.

Per definire una *box* invece servono $2 \cdot n$ valori: è sufficiente considerare le coordinate di due qualsiasi vertici opposti: ad esempio per definire un box in 3 dimensioni bastano 6 vertici, due per ogni coordinata (cioè il massimo e il minimo per ogni direzione). La complessità dell'algoritmo di calcolo della *MMB* per un insieme di N punti è lineare con N, cioè $O(N)$, infatti basta trovare i valori minimo e massimo per ogni coordinata. Si nota dunque che MMB oltre ad essere di facile compresione, risulta molto facile da creare.

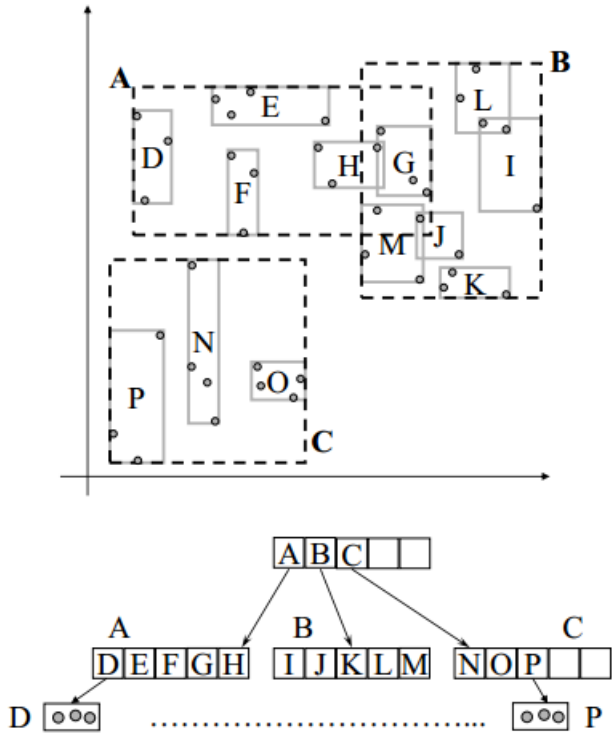
Vediamo ora un veloce confronto tra un *B+tree* e un *R-tree* (Tab. 1)

Tabella 1:

B+tree	**R-tree**
Albero bilanciato e paginato	Albero bilanciato e paginato
I dati si trovano nelle foglie	I dati si trovano nelle foglie
Le foglie sono ordinate	Non esiste ordine tra i dati
Organizza i dati in intervall 1-D: gli intervalli non si sovrappongono	Organizza i dati in intervalli n-D (MBB): gli intervalli si sovrappongono (caratteristica dello spazio n-D)
Il principio viene applicato ricorsivamente verso l'alto	Il principio viene applicato ricorsivamente verso l'alto
La ricerca puntuale segue un solo percorso dalla radice alle foglie	La ricerca puntuale segue più percorsi dalla radice alle foglie

Vediamo di seguito come un R-tree organizza i propri dati, si notino i vari BMM che si sono creati e l'albero che poi viene generato (Fig. 25)

Figura 25:



I *nodi foglia* di un R-tree contengono *entry* del tipo (*key*, *RID*), dove la *key* contiene le coordinate del record: in realtà R-tree può anche gesitre oggetti n-dimensionali con un'estensione spaziale e quindi con $key = MMB$ (ad esempio potrebbe usare come *key* un MMB che rappresenta un lago). I *nodi interni*, invece, contengono *entry* del tipo (*MBB*, *PID*), dove la MMB contiene le coordinate della MMB delle entry dei figlie e *PID* il puntatore alla pagina dell'MMB. Uniformando, ogni nodo contiene delle *entry* (*key*, *ptr*) dove key indica un valore "spaziale". Inoltre ogni nodo contiene un numero di m entry che può variare tra c e C ($c \leq m \leq C$) in cui $c \leq \frac{C}{2}$ è un parametro di utilizzazione, mentre C dipende dal valore della dimensione n dello spazio e dalla dimensione della pagina: la radice è l'unico nodo che può violare il vincolo di minima utilizzazione ed avere anche due sole entry per lo stesso motivo per cui accadeva in B+tree.

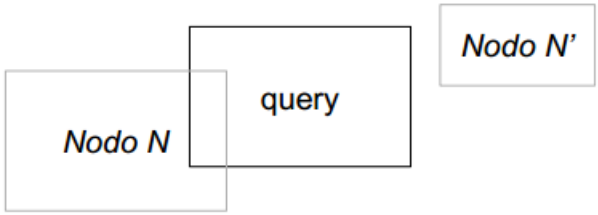
Osserviamo infine la differenza sostanziale tra la ricerca del minor (iper-)rettangolo che contiene N punti, e la ricerca della minore *MMB* che riesca a contenerli. Per l'(iper-)rettangolo, in uno spazio a 2 dimensioni gli algoritmi esistenti hanno tempo $O(N^2)$, in uno spazio a 3 dimensioni $O(N^3)$ e in spazi a dimensioni maggiori si conoscono solo algoritmi esponenziali nel numero di punti. Tutt'altro caso invece è trovare la MMB che contenga N punti: infatti il

semplice algoritmo per calcolarla ha tempo $O(N \cdot d)$ dove d indica la dimensione sul quale si sta lavorando. L'algoritmo semplicemente cerca il minimo fra i minimi e il massimo fra i massimi in ogni coordinata.

1.7.1 Ricerca

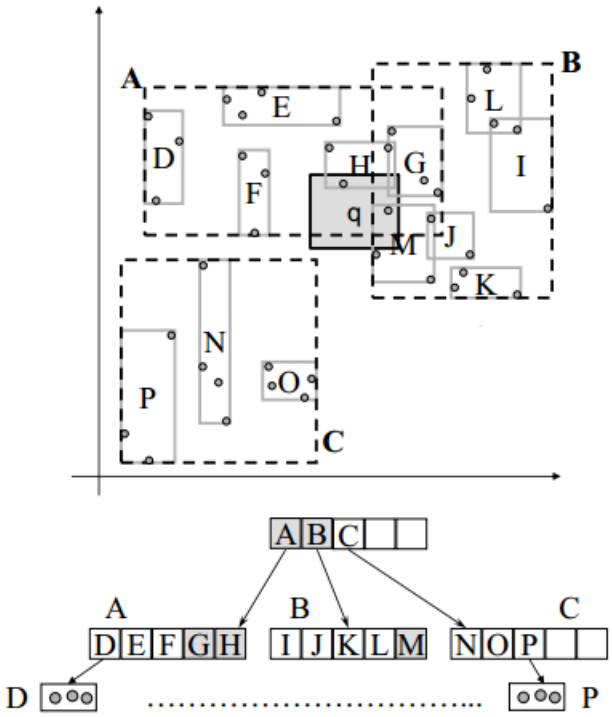
Vediamo ora come avviene la ricerca all'interno di un R-tree. Immaginiamo una ricerca per *window query*: occore, quindi, trovare tutti i dati inclusi in un prodotto di n intervalli (ovvero in una box). Tali dati possono trovarsi solamente in nodi la cui MBB ha *sovrapposizione* con la query. Si veda la seguente figura (Fig. 26): il nodo N' non può contenere record che soddisfino la query. A causa dell'innestamento gerarchico di bounding box, se una foglia ha intersezione non vuota con la *window query*, allora anche il padre, ed il nonno ed ogni antenato della foglia avrà intersezione con la query. Quando trovo invece una box che ha intersezione nulla con la window query allora sono sicuro che esplorando quel'albero non troverò alcuna soluzione e dunque posso scartarlo. Come sempre si scopre solo al livello delle foglie quali dati sono presenti nel risultato alla *window query* e se effettivamente né esistano, di conseguenza se la nostra window query ha intersezione non vuota con più di una foglia, allora bisogna esplorarle tutte. Conseguentemente, siccome l'algoritmo procede dalla radice verso le foglie, se nel corso della ricerca la query ha intersezione non vuota con più di un nodo interno (non foglia), bisogna per forza applicare l'algoritmo di *backtracking*, poichè, infatti, se un nodo interno porterà a un sotto-insieme di foglie, l'altro nodo ne conterrà di diverse ma di cui ugualmente la visita è necessaria.

Figura 26:



Si veda come, a livello generico, nella seguente figura (Fig. 27), l'algoritmo procede. Siccome i dati si trovano nel box q, il quale inizialmente ha un intersezione non vuota solo con A e B (e non con C) e al livello successivo solo con i box G, H e M, i dati del risultato si troveranno solo all'interno di questi Box. Come si vede i risultati sono effettivamente presenti solo nella foglia H e nella foglia M, quindi la foglia G è stata visita 'inutilmente' ma era necessario farlo.

Figura 27:



Riprendiamo in mano gli algoritmi di ricerca che avevamo discusso quando abbiamo parlato dei GiST, ed applichiamoli al'R-tree.

Ricordiamo in velocità a cosa serve il GiST: è un modo generale per implementare alberi in cui ogni nodo è una entry del tipo (chiave, puntatore), in cui deve valere la *monotonicità* dei predicati ovvero che se il predicato è valido per un determinato nodo, deve essere valido anche per il padre.

CONSISTENT `Consistent(E,q)`

- **Input**: *Entry E=(p,ptr)* e *predicato di ricerca q*
- **Output**: `if (p & q) == false then false else true`

Sia p che q sono in questo caso (iper-)rettangoli. *Consistent* restituisce *true* se e solo se p e q hanno sovrapposizione (*overlap*) non nulla. *Consistent* è indifferente alla "forma" di q (ovvero non è richiesto che q sia un MMB mentre p lo è poichè stiamo trattando un R-tree), il quale può anche essere usato per altre query (come per *range* or *nearest neighbour*) e potrebbe avere anche una forma curvilinea o sferica. Ne consegue che la ricerca può seguire più percorsi nell'albero sia perchè la *query* non è puntuale (è una window query) sia perchè le MMB si sovrappongono (*overlapping regions*).

È bene aprire però una parentesi sul metodo Consistent. Non è detto che la ricerca sia conservativa, ovvero non è obbligatorio che i risultati effettivi siano "esatti". Esistono due casi di questo tipo: il primo, ad esempio, accade quando l'indice (poichè ricordiamo che stiamo parlando di indici quando trattiamo R-tree) restituisce informazioni in più rispetto ai dati che effettivamente volevamo, cioè che restituisce sicuramente anche il risultato esatto, ma in più dà informazioni errate: questo è dovuto al comportamento conservativo che potremmo applicare al metodo Consistent. Ad esempio se applichiamo un Consistent che risponda sempre *true* esso risulterà "troppo" conservativo (poichè nessun nodo risponde falso, neanche le foglie di conseguenza non è possibile che un nodo restitusica falso mentre il suo figlio avrebbe potuto rispondere vero): ciò porta ad avere nel risultato finale tutti i dati a disposizione di cui sicuramente un sotto-insieme è il 'vero' risultato della nostra query, e che come abbiamo detto, in una caso di questo tipo vengono fornite informazioni in più (tutti i restanti i dati). Tutto ciò comporta ad una grande inefficienza poichè si richiede di accedere a tutti i dati (come se l'indice non ci fosse), ed in questi casi si parla "*falsi positivi*".

Dall'altra parte invece i risultati potrebbero non essere "esatti" perchè utilizziamo un Consistent "troppo poco conservativo", cioè quando il Consistent restituisce falso anche se in realtà dovrebbe restituire vero e in questi casi si parla di "*falsi negativi*". Immaginiamo di avere un Consistent che restituisce sempre *false*: allora il risultato di qualsiasi query sarà sempre un'insieme vuoto poichè già alla radice l'algoritmo di ricerca si blocca. Ovviamente anche in questo caso abbiamo un risultato "errato" ma in maniera opposta rispetto al caso prima trattato: infatti ora si ottengono meno informazioni del "vero" risultato (in questo caso proprio nessuna) ma il calcolo del risultato è estremamente veloce (ci si ferma al primo nodo). Come sempre dunque esistono dei *trade-off*: rispondere falso quando in realtà risulta vero dà una maggiore efficienza poichè si taglia tutto il sotto-albero del nodo ma restituisce risultati in meno di quelli effettivi, mentre rispondere vero quando in realtà risulta falso aggrava la performance poichè richiede di arrivare fino alle foglie per scoprire effettivamente la falsità del predicato.

Solitamente i *dbms* adottano una tecnica chiamata *filter and refine*: viene cioè utilizzato un Consistent "molto conservativo" in cui si preferisce avere informazioni in eccesso ma fra le quali è presente effettivamente il risultato. A queste ultime viene dunque applicato un ulteriore algoritmo di "raffinamento" che riesce ad eliminare i falsi positivi che si erano creati. Questo algoritmo di "raffinamento" è solitamente molto costoso e applicarlo direttamente a tutto l'insieme di dati potrebbe richiedere un tempo molto grande, ecco dunque che se invece prima si "filtra" l'insieme dei dati attraverso l'indice e solo a questi ultimi viene applicato l'algoritmo, la performance generale riesce ad aumentare notevolmente riuscendo a dare in output il "vero" risultato della query.

Occorre, infine, specificare i *key methods* Union, (Compress, Decompress), Penalty e PickSplit ovvero i *metodi* per la costruzione dell'albero (insieme al Consistent di cui abbiamo già parlato). Esistono diverse "varianti" di R-tree,

ognuna delle quali si differenzia dalle altre per uno o più modi di implementare tali scelte. Vedremo la realizzazione della versione orginale di R-tree e discuteremo alcune varianti, fra le quali una delle più comuni R*-tree (Beckmann et al, 1990). Notiamo che per *Union* esiste una sola implementazione, mentre a seconda di come viene implementata Penalty e PickSplit si costruiscono varianti del R-tree. L'arbitrarietà delle scelte è causata dalla dimensione dello spazio maggiore di uno, in particolare è dovuta al fatto che non esiste un ordine totale fra gli elementi.

UNION `Union(P)`

- **Input**: Insieme di entry $P = \{(p_1, ptr_1), \ldots, (p_n, ptr_n)\}$
- **Output**: Un predicato r che vale per tutte le tuple accessibili tramite uno dei puntatori delle entry

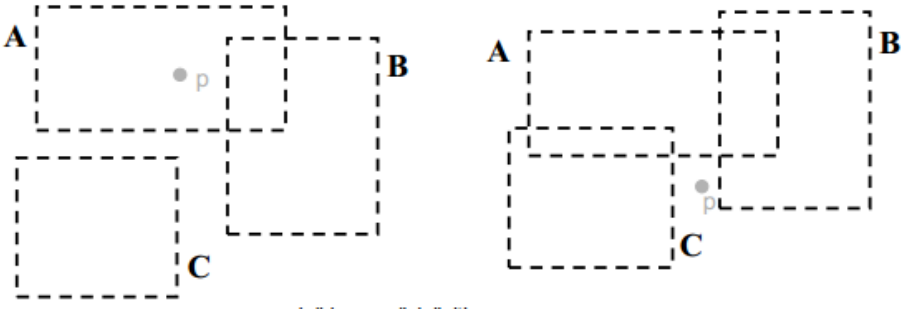
Sia il predicato r che i predicati p_j sono (iper-)rettangoli. Si restituisce la *MBB* contenente tutti i predicati p_j. È sufficiente calcolare il valore minimo e massimo su ogni coordinata (che ricordiamo essere $O(n \cdot d)$).

PENALTY `Penalty(E1, E2)`

- **Input**: Entries E_1=(p_1, ptr_1) e E_2=(p_2, ptr_2)
- **Output**: Un valore di "penalità" che risulta dall'inserire E_2 nel sottoalbero con radice E_1

Si veda l'esempio in figura (Fig. 28) e ci si domandi quale sia il nodo migliore per inserire il punto p.

Figura 28:



Se il punto p si trova all'interno di E_1, la penalità è zero. Altrimenti la penalità è data dall'incremento di volume (area) della *MBB*. Qui esiste una prima differenza con R*-tree: infatti se mentre nei nodi interni applica nuovamente la minimizzazione del volume, in caso di foglie applica come penalty l'incremento di sovrapposizione con le altre entry.

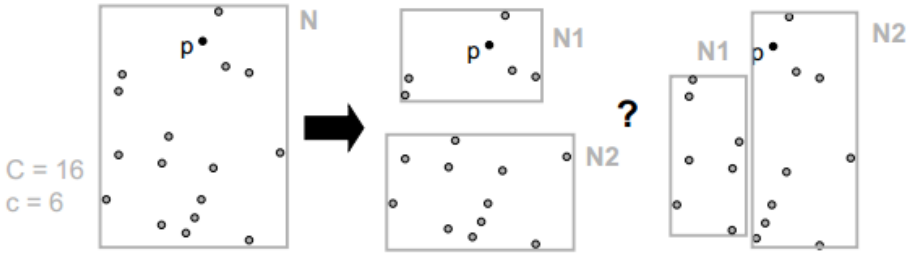
Entrambi i criteri cercano di ottenere un albero con migliori prestazioni: se da una parte ho un *volume grande* che porta all'aumento delle probabilità di visitare un nodo durante una query, dall'altra ho un *overlap grande* il quale aumenta il numero dei nodi visitati da una query. Di conseguenza minimizzare il volume vuol dire avere meno probabilità che la *window query* e la MMB si intersichino, mentre minimizzare l'overlapping delle foglie ha come scopo diminuire la probabilità di dover accedere a più foglie per controllare uno stesso elemento.

PICKSPLIT `PickSplit(P)`

- **Input**: Insieme di C+1 entry
- **Output**: Due insiemi di entry, P_1 e P_2 di cardinalità $\geq c$

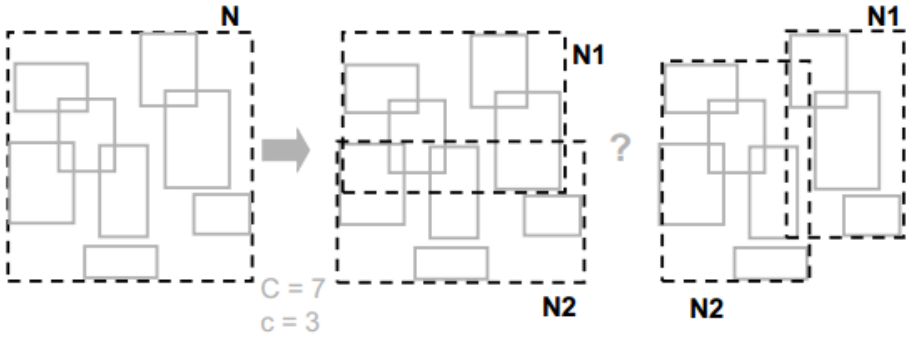
Si veda l'esempio mostrato in figura in cui si ha capacità massima $C = 16$ e capacità minima $c = 6$ e in cui aggiungendo un nuovo oggetto, la foglia vada in overflow (Fig. 29). Esiste un problema intrinseco: in che modo divido la foglia? Il dubbio nasce dalle diverse partizioni possibili che si possono creare.

Figura 29:



Cercare la divisione che porta a *minimizzare* la somma dei volumi due nodi è un problema NP-hard, ecco che si usano delle euristiche per risolvere il problema. Se mentre per le foglie è sempre possibile ottenere una divisione senza overlapping, la cosa si complica ulteriorimente nei nodi superiori: in particolare non è garantito che si trovi uno split ad *overlap* nullo, cioè è possibile che lo split in due nodi abbia un'interesezione. Si veda d'esempio la seguente figura (Fig. 30).

Figura 30:



Il criterio adottato da R*-tree è più complicato e tiene in considerazione sia il volume dei nodi che il loro overlap ed il perimetro (cercando dunque di minimizzare perimetro e overlap nelle foglie, e volume nei nodi superiori). Inoltre R*-tree supporta la re-distribuzione sia in *overflow* che in *underflow*. Tutte queste scelte sono effettuate in maniera *euristica*, in quanto la loro efficienza viene validata unicamente in modo sperimentale. Con queste specifiche si ottengono (lievi) miglioramenti nelle prestazioni in inserimento, occupazione e ricerca.

1.7.2 R-tree vs B+tree

Facciamo un breve confronto fra i due alberi per quanto riguarda le ricerche, supponendo di avere sempre query applicate ad attributi A e B e sul quale siano stati costruiti un indice R-tree su (A,B) e un indice B+tree su (A,B) (in questo ordine nel caso di B+tree).

- In caso di ricerca puntuale solo sull'attributo A, il B+tree vince sulle performance, poichè sicuramente chiede di accedere a una sola foglia (al più si prosegue per concatenazione di foglie), mentre con R-tree potrebbe essere che il valore si trovi in una interesezione di MMB il che richiede l'accesso a più foglie (con meccanismo di backtracking)
- Caso di intervallo solo su A: ancora una volta B+tree vince, poichè si accede direttamente alle foglie che risultano ordinate e si prosegue grazie alla concatenazione di esse. Invece su R-tree nuovamente, a causa della possibile intersezione di overlapping si potrebbe dover accedere a più foglie con meccanismi di overlapping.
- Ricerca puntuale su A e intervallo su B: B+tree è ancora da padrone, è molto simle alla ricerca puntuale solo su A. Si noti che nello spazio a due dimensioni una ricerca di questo tipo è raffigurata da un segmento, e ancora una volta R-tree a causa di interesezione potrebbe dover accedere a più foglie.
- Ricerca intervallo su A e intervallo su B: R-tree vince, è proprio il motivo per cui è stato costruito l'R-tree (rappresenta la window query).

- Ricerca puntuale su B: con B+tree non è possibile farla, mentre con R-tree è come effettuare una ricerca puntuale su A
- Intervallo su B: con B+tree non risulta possibile, con R-tree ricadiamo nel caso di ricerca per intervallo su solo A.