Tecnologie delle Basi di Dati M

Antonio Davide Calì

March 28, 2014

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

Contents

Part I
Strutture di Indicizzazione

Nel capitolo precedente abbiamo introdotto l'organizzazione delle pagine (e record) attraverso file, le quali indubbiamente portano alcuni vantaggi quali inserimenti molto veloci (nel caso di *heap file)* e ricerche mediamente rapide per quasi tutte le operazioni (in caso di *sequential file*). Entrambe però hanno alcuni svantaggi, infatti la ricerca negli *heap file* risulta molto lenta, mentre la ricerca sui *sequential file* risulta efficiente solo se è effettuata sull'attributo di ordinamento (che richiede inoltre periodiche riorganizzazioni).

Si può dunque far di meglio?

1 Strutture di indicizzazione

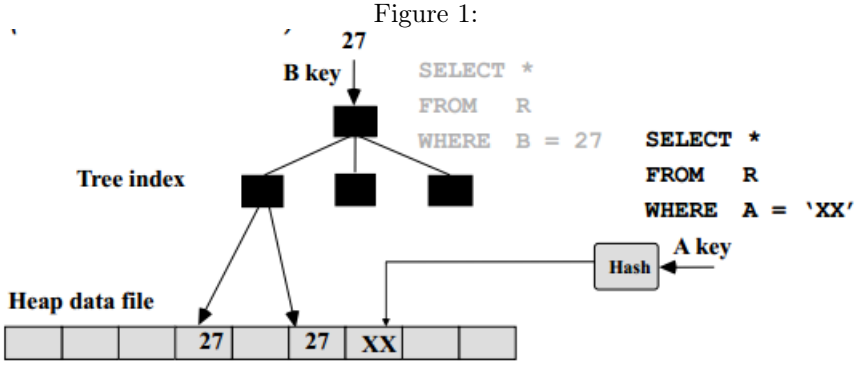
Le strutture di indicizzazione sono strutture *ausiliarie* che permettono di recuperare velocemente i **RID** (Record ID) dei crecord che soddisfano una certa condizione. Ogni **indice** facilita la ricerca di una diversa condizione (*chiave di ricerca*) ed è (sostanzialmente) formato da una collezione di coppie *<chiave,RID>* detta **entry**. Lo scopo dell'indicie è quello di velocizzare il recupero delle *entry* il cui valore della chiave soddisfa la condizione. Il vantaggio introdotto dagli *indici* è dovuto alla grandezza ridotta delle *entry* rispetto alla grandezza maggiore dei *record*, questo permetterà dunque di organizzare gli indici su disco in maniera "furba" per riuscire ad accedere ad essi in modo sequenziale. I dati vengono comunque memorizzati come già visto precedentemente (*heap/sequential file*).

Gli indici sono indubbiamente strutture molto comode, e ci si potrebbe chiedere perchè non applicarli a tutti gli attributi di tutte le relazioni. Mantenere gli indici ha un costo elevato in termini di prestazioni, in quanto ad ogni modifica/aggiunta/rimozione di un *record* in una tabella, bisogna aggiornare tutti gli indici relativi. Bisogna inoltre ricordare che per quanto la grandezza degli indici sia molto inferiore a quella delle relazioni, essi comunque occupano

spazio su disco. Non è dunque pensabile di mantenere un indice per ogni attributo ed è infatti compito dell'*amministratore* del database decidere per quali attributi creare gli indici.

La costruzione di indici su una relazione (tabella) serve per dare modalità alternative, chiamate **cammini di accesso**, per localizzare e accedere velocemente ai dati di interesse.

Figure 1:



In figura (Fig. 1) vengono mostrati effettivamente due possibili cammini di accesso, il primo attraverso un albero e il secondo attraverso una funzione hash (li vedremo entrambi più avanti).

Si usa comunemente il termine (valore di) **chiave** (di ricerca) per indicare il valore di un campo usato per la selezione dei record (Esempio. B è una chiave).

2 Indici

Quale è il principio di base degli indici?

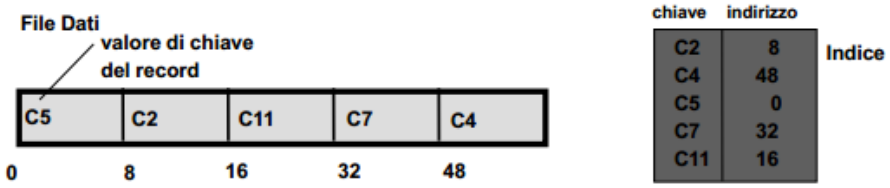
Sostanzialmente un indice è visto come un *insieme di coppie* (**entry**) del tipo $(\mathbf{k_i}, \mathbf{p_i})$ dove:

- $\mathbf{k_i}$ è un *valore di chiave* del campo su cui l'indice è costruito
- $\mathbf{p_i}$ è un *puntatore ai record* (eventualmente uno solo se l'unico) con il valore di chiave k_i(nei *dbms* è quindi il **RID** o al limite un **PID**).

La entry è dunque formata da una coppia <chiave,RID> è fatta da un valore presente nell'attributo per il quale stiamo creando l'indice (chiave) e da uno o più RID poichè è possibile che ci siano più *record* che abbiano lo stesso valore dell'attributo per il quale l'indice viene creato (chiave).

Il vantaggio di usare un indice nasce dal fatto che la chiave è solo parte dell'informazione contenuta in un record, pertanto *l'indice occupa uno spazio molto minore rispetto al file dati.*

Figure 2:

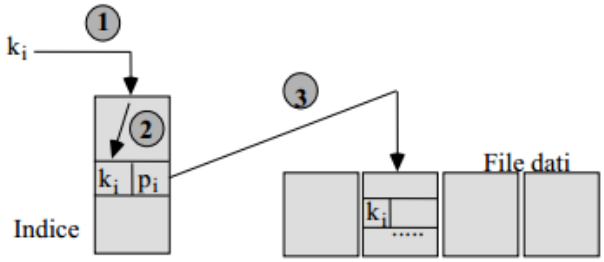


I diversi indici differiscono essenzialmente nel modo con cui vengono organizzate le coppie $(\mathbf{k_i,p_i})$.

Si consideri ora un indce su chiave primaria usato per ricercare il record con chiave k_i. Lo schema generale per accedere è dato da:

1. Accede all'indice
2. Cercare la coppia (k_i,p_i)
3. Accedere alla pagina contenente i dati di interesse

Figure 3:



2.1 Tipi di Indici

Esistono diverse tipologie di *indici*. Una prima distinzione la si ha tra:

- **Indici ordinati**: i valori di chiave k_i vengono mantenuti ordinati in modo da poter essere reperiti più efficientemente attraverso una ricerca binaria (di contro devo appunto mantenere ordinato l'indice).
- **Indici hash**: si usa una funzione hash per determina la posizione dei valori di chiave k_i. Il vantaggio lo si trova in una *ricerca per chiave* in quanto con una sola operazione ottengo direttamente il **RID**, di contro però forniscono pessime prestazioni nelle *ricerche per intervallo* poichè la funzione hash memorizza due valori molto vicini (esempio: case e casi) in ordine completamente sparso, dunque per accedervi devo effettivamente chiamare h(case) e h(casi). *h denota la funzione hash.*

2.1.1 Nomenclatura

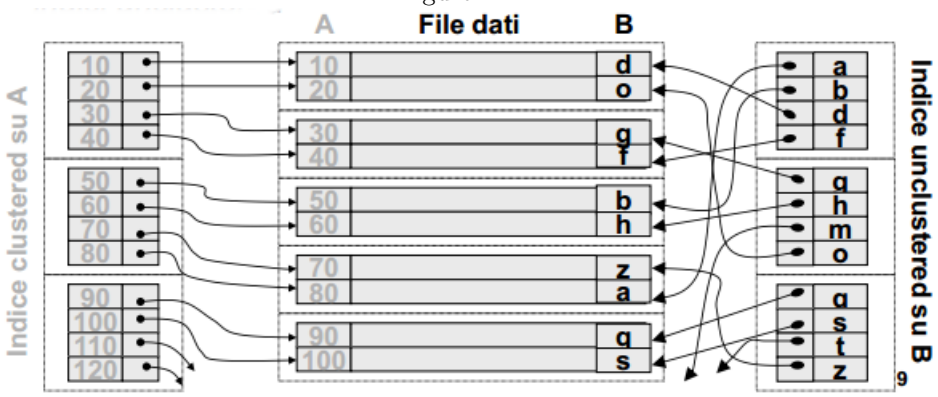
La *terminologia* che andremo ad introdurre **non** è standard anche se è la più diffusa (infatti alcuni non fanno distinzione fra i primary/clustered e secondary/unclustered).

- **Clustered vs unclustered**
- **Primary vs secondary**
- **Single-level vs multi-level**
- **Dense vs sparse**

Clustered e Unclustered Un indice è detto **clustered** se è costrutito sul campo su cui i *record* nel file dati sono mantenuti ordinati (cioè se l'ordinamento dei *record* sul file e dell'indice è lo stesso). L'indice è detto **unclustered** altrimenti.

Ovviamente si può costruire al massimo un indice clustered per relazione (poichè il file può essere ordinato solo in un modo e solo un indice può essere costruito ottenendo lo stesso ordinamento), mentre possono essere costrutiti un numero arbitrario di indici unclustered.

Figure 4:

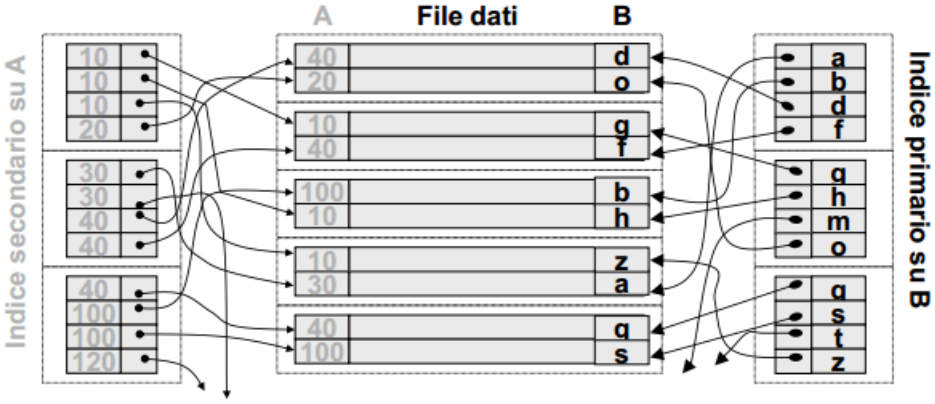


Nonostante l'ordinamento dei *record* dei file dati e dell'*indice* risulti uguale negli indici **clustered**, essi ricoprono un ruolo essenziale dovuti al fatto che la loro dimensione risulta essere molto ridotta rispetto al file dati.

Primary e Secondary Un indice è detto **primary** (primario) se è costruito su un campo a valori non ripetuti (chiave candidata, attributo *UNIQUE*) e dunque ad ogni valore di chiave è associato un solo RID. L'indice è detto **secondary** altrimenti e dunque sono possibili più RID per un unico valore di chiave.

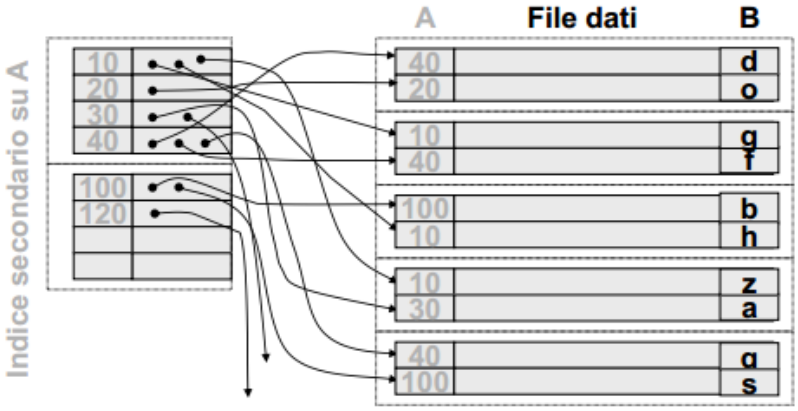
In figura (Fig. 5) si supponga attributo B come attributo UNIQUE.

Figure 5:



Negli indici **secondary** per evitare di replicare inutilmente valori di chiave, la soluzione più comunemente adottata consiste nel *raggruppare tutte le coppie con lo stesso valore di chiave in una lista di puntatori* (Fig. 6).

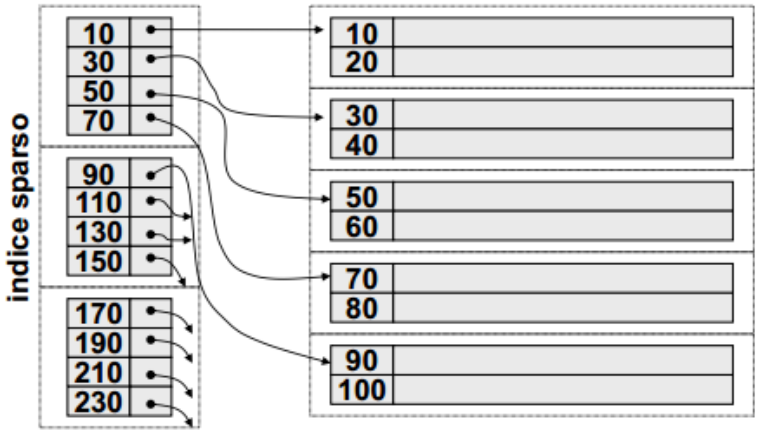
Figure 6:



Dense e Sparse Un indice si dice **dense** (denso) se il numero di entry dell'indice è pari al numero di *record* del file dati. L'indice si disce **sparse** (sparso) in caso il numero di entry risulti minore (tipicamente uno per pagina dati). L'indice sparso è una *soluzione applicabile con indici clustered,* e il vantaggio evidente risulta essere la dimensione sempre più ridotta dell'indice stesso. Lo svantaggio da pagare è presente in caso dell'assenza di un determinato valore di chiave. Si veda l'esempio in figura (Fig.), se si vuole cercare la chiave **25** (non presente), con un indice sparso sono obbligato a leggere l'intera pagina associata

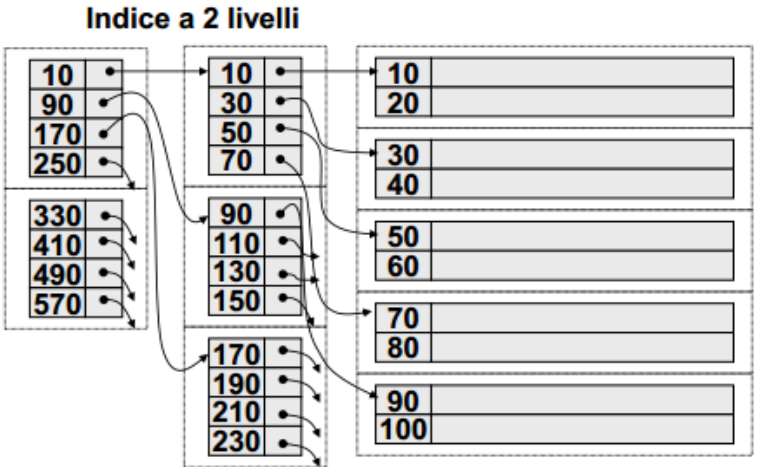
all'indice **10** per accorgermi che l'entry **25** non esiste. Invece se l'indice fosse stato denso, ci si accorgeva subito (senza dover leggere la pagina) che il valore non era presente.

Figure 7:



Single-level e Multi-level Un indice si dice **single-level** se esso indicizza direttamente un *file dati*. È tuttavia possibile "indicizzare indici" creando dunque una **struttura multi-livello (albero)** utilizzando indici *sparsi* (altrimenti creerei indici identici).

Figure 8:



N.B. Il primo livello è l'indice che indicizza indici, il secondo livello è l'indice che indicizza il *file dati (Fig. 8)*.

Per ragioni di efficienza solitamente gli indici sono **multi-livello** (alberi), poichè, ad esempio, se continuo ad indicizzare indici fino a che non mi ritrovo ad avere un indice con una sola entry ho costruito esattamente un albero. In questo caso posso utilizzare la *ricerca binaria* applicata ad un albero piuttosto che la *ricerca binaria* applicata ai *file dati*. Ad ogni livello di indice si legge una sola pagina, dunque se si hanno 5 indici si leggono solo 5 pagine.

2.1.2 Creazione Indici SQL

In SQL la definizione di indici avvine mediante lo statemente CREATE INDEX (non standardizzato).

Alcuni esempi in DB2:

- `CREATE INDEX VotoIDX ON Esami(Voto DESC)` crea un indice di nome VotoIDX applicandolo all'attributo Voto della tabella Esami (la specifica DESC per dire che voglio ordinarlo in maniera discendente, ASC è la specifica di default). Essendo Voto un attributo non UNIQUE l'indice sarà *unclustered*

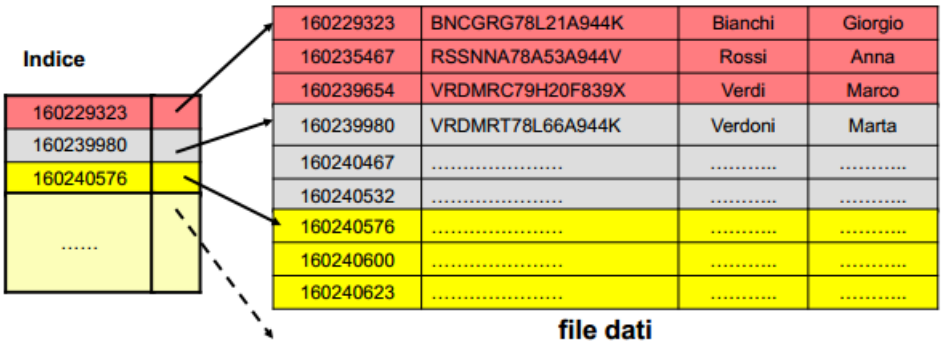
- `CREATE UNIQUE INDEX MatrIDX ON Studenti(Matricola)` crea un indice MatrIDX su attributo Matricola della tabella Studenti. La specifica *UNIQUE* è facoltativa, il sistema si accorge automaticamente di star lavorando su attributi chiave. L'indice è di tipo *clustered*. NB Non è possibile specificare UNIQUE su un indice applicato a un attributo che UNIQUE non è.

- `CREATE INDEX VotoIDX ON Esami(Voto DESC) CLUSTER` crea un indice uguale al primo indicato. Aggiungendo però la specifica CLUSTER obbligherà il sistema a ordinare la **tabella** Esami sull'attributo Voto in maniera discendente. Ogni volta che viene aggiunta la specifica CLUSTER a un indice, il sistema riordina la tabella associata, "declusterizzando" i precedenti indici che erano stati segnati CLUSTER (vengono "declusterizzati" nel senso che l'indice automaticamente non rispecchierà più l'ordine del file).

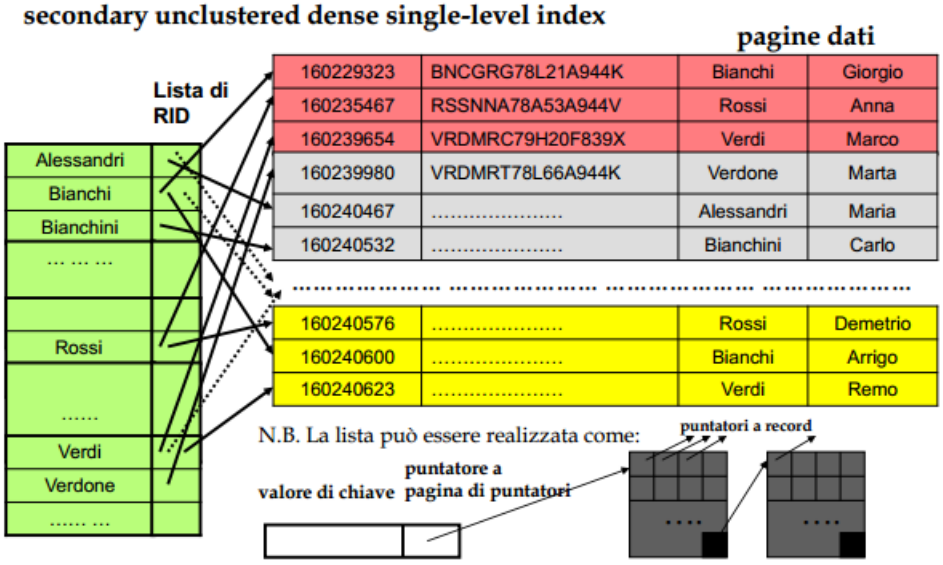
- `CREATE INDEX Anagrafica ON Persone(Cognome,Nome)` crea un indice multi-attributo (ordinato quindi per cognome-nome).

Alcuni esempi

Figure 9:
primary clustered sparse single-level index



secondary unclustered dense single-level index



2.2 B-Tree

Ci si potrebbe chiedere se possiamo "adattare" alla memoria secondaria gli alberi di ricerca pensati per la memoria centrale.

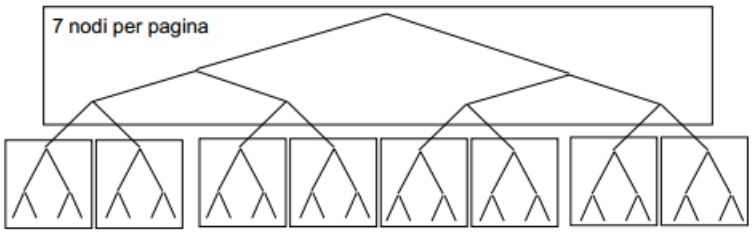
Purtroppo questo non è possibile in quanto sono richiesti alcuni requisiti che in memoria secondaria non vengono rispettati:

- Bilanciamento (prestazioni nel caso peggiore).

- Paginazione (letture da disco). Il problema fulcro riguarda il rispetto di questo requisito in quanto gli *indici* devono essere paginati, e non è facile spostare poi gli indici da memoria principale a memoria secondaria e viceversa.
- Utilizzazione minima delle pagine (dimensioni). Bisogna evitare di creare spazio inutilizzato.
- Effiencienza in aggiormento

Solitamente infatti, in memoria centrale, gli alberi sono tipicamente binari (*AVL tree, red-black tre*), il cui numero di nodi è molto alto (e dunque la visita richiede molti accessi) i quali vengono inclusi all'interno di una pagina (ad esempio 7 nodi per pagina). Se come costo dell'algoritmo prendessi la visita di un nodo, avere molti nodi implicherebbe un costo eccessivo. Una buona soluzione sarebbe quindi quella di avere un numero di nodi non proporzionale al numero di tuple (*record*) ma bensì al numero di *pagine*. Ricordiamo che il nostro scopo è riuscire a paginare in maniera efficiente gli alberi per la costruzione di indici.

Figure 10:



Con un tale schema di albero binario purtroppo si crea in efficienza, in quanto si creano complicazioni sull'algoritmo di bilanciamento e inoltre non viene data alcuna garanzia di avere occupazione minima delle pagine. Bisogna quindi trovare una soluzione specifica.

2.2.1 Definizione B-Tree

Ci serve dunque una struttura dati ad albero che mantenga i *dati ordinati* e i *nodi bilanciati* permettendo operazioni di inserimento, cancellazione e ricerca in *tempo ammortizzati logaritmicamente*.

Un **B-tree** è un *albero a più vie perfettamente bilanciato organizzato a nodi che corrispondono a pagine*. Dire che l'albero sia *a più vie* significa che il fattore di diramazione è≥ 2, mentre dire che sia *perfettamente bilanciato* implica che *il percorso dalla radice ad una qualsiasi foglia ha la stessa lunghezza* (altezza dell'albero). Ogni nodo contiene un numero di *entry* **m** che può variare tra **d** e **2d** $d \leq m \leq 2d$ (**d**=***ordine*** dell'albero), cioè in ogni nodo sono memorizzate più di una *entry* (il numero di nodi non è più proporzionale al numero di tuple).

Il numero di nodi figli di un nodo è pari a **m+1** (può dunque variare tra **d+1** e **2d+1**). Questo implica un *fan-out* elevato e quindi altezza limitata, un costo di ricerca (molto) basso e dimensioni ridotte. Il costo per accedere a una foglia è sempre pari all'*altezza* dell'albero.

NB. La radice è l'unico nodo che può violare il vincolo di minima utilizzazione e avere anche una sola *entry* (dunque per la radice il numero **m** è compreso tra $1 \leq m \leq 2d$).

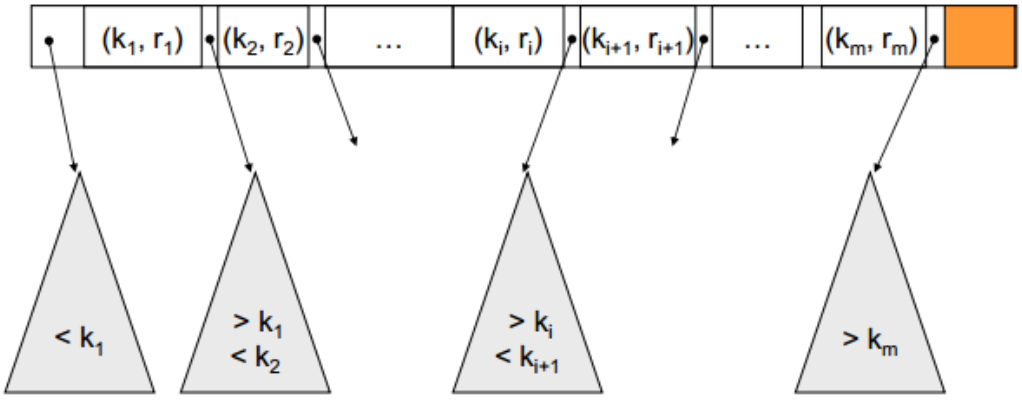
Ogni ricerca *segue un unico percorso dalla radice ad una foglia* (*costo* $\leq$ *altezza* quindi operazioni ammortizzate logaritmicamente). Il *bilanciamento* è garantito dalle operazione di **inserimento** e **cancellazione** dei record (la "chiave" per mantenere l'albero bilanciato risiede nel fatto che il numero di *figli* è variabile), dunque le operazioni di diramazione di un nodo avvengo verso l'alto e non verso il basso (cioè in caso di sbilanciamento, una foglia può essere portata ad essere un figlio alternativo del nodo *antenato*). L'occupazione minima è del 50% (ad eccezione della radice), mentre l'occupazione tipica è $> 66\%$.

2.2.2 Formato dei nodi

Il formato dei nodi differisce a seconda che il nodo sia *interno* o sia una *foglia*.

Nodi Interni I nodi interni hanno il formato mostrato in figura (Fig. 11) in cui $k_1 < k_2 < \ldots < k_m$.

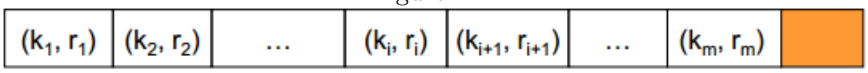
Figure 11:



Come si evince dalla figura i nodi interni sono dunque composti sia da **entry** (k_i,r_i) sia da *puntatori* ai nodi *figli*.

Foglie Le foglie hanno un formato (Fig. 12) simile a quello dei nodi interni in cui $k_1 < k_2 < \ldots < k_m$.

Figure 12:



Vista la mancanza dei puntatori ai *sotto-alberi* tipicamente le foglie contengono più entry dei nodi interni.

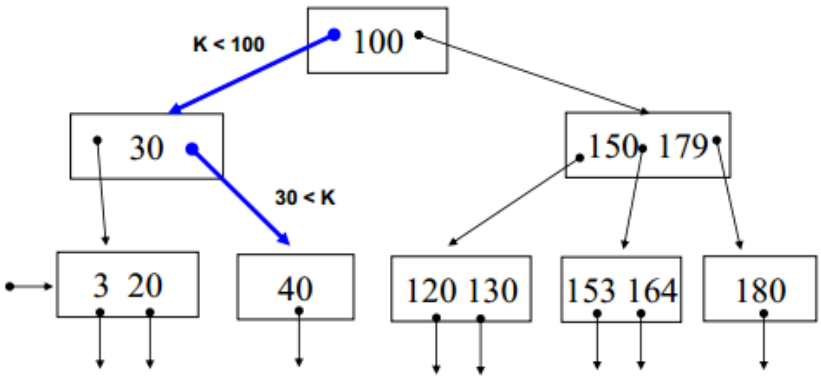
2.2.3 Algoritmo di Ricerca

L'algoritmo di ricerca in un **B-tree** è relativamente semplice ma altrettanto efficace.

1. Si parte dalla radice dell'albero (generalmente le radici di tutti i *b-tree* sono mantenute in memoria centrale per questioni di efficienza).
2. Si cerca la chiave k_i tra quelle del nodo corrente
   - Se è presente: fine (Result: **trovato**)
   - Se non è presente e siamo in una *foglia*: fine (Result: **non trovato**)
   - Se non è presente e siamo in un nodo interno, sostituisci il nodo corrente in memoria con il i-esimo nodo figlio dove $k_{i-1} < k < k_i$

Esempio

Figure 13:



Cerchiamo la chiave k=40

1. 40<100 ⇒seguiamo il figlio sinistro
2. 40>30 ⇒seguiamo il figlio destro
3. 40 è presente nelle entry del nodo corrente: fine (Result: **trovato**)

Costo della ricerca Ogni nodo caricato in memoria viene sostituito (in caso) da un unico figlio (dunque l'occupazione in memoria è ottima). Nel caso peggiore bisogna arrivare ad una foglia. Il costo è comunque sempre minore uguale all'altezza dell'albero: $costo \leq altezza\,dell'albero - 1 + 1$ dove

- -1 poichè la radice è già pre-caricata in memoria centrale
- $+1$ perchè una volta trovata l'**entry** bisogna accedere al *file dati* (potrebbe essere non necessario se l'indice invece che contenere le **entry** contenesse direttamente il *file dati*).

Ne consegue che per sapere il costo della ricerca *occorre saper calcolare l'altezza di un B-tree* di ordine **d**.

2.2.4 Numero nodi di un B-tree

Il numero dei nodi (**b**) presenti in un B-tree varia, ovviamente, tra in un intervallo compreso fra $b_{min} \leq b \leq b_{max}$. Non ci resta dunque che calcolare $\mathrm{b_{min}}$ e $\mathrm{b_{max}}$.

Numero Minimo di Nodi Il numero minimo di nodi si ha quando tutti i nodi (ad eccezione della *radice*) sono pieni a metà, ovvero $m = d$ (numero di entry uguale all'ordine dell'albero) e quando la radice ha esattamente una sola *entry* (il che vuol dire che ha due soli nodi figli).

Ogni nodo interno ha quindi **d+1** figli e dunque il numero di nodi è dato da

$$b_{min} = 1 + 2\sum_{I=0}^{h-2}(d+1)^I = 1 + 2\frac{(d+1)^{h-1} - 1}{d}$$

e il corrispondente numero di *entry* è dunque di

$$N_{min} = 1 + d \cdot (b_{min} - 1) = 2(d+1)^{h-1} - 1$$

Numero Massimo di nodi Il numero massimo di nodi si ha quando tutti i nodi sono pieni (compresa la radice) ovvero $m = 2d$ (numero di entry uguale a 2d).

Ogni nodo interno ha quindi **2d+1** *figli* e dunque il numero di nodi è dato da

$$b_{max} = \sum_{I=0}^{h-1}(2d+1)^I = \frac{(2d+1)^h - 1}{2d}$$

e il corrispondente numero di *entry* è dunque di

$$N_{max} = 2d \cdot b_{max} = (2d+1)^h - 1$$

2.2.5 Altezza B-tree

Dato il numero di *entry* N (numero noto) è possibile dunque calcolare l'altezza del **B-tree**

$$N_{min} \leq N \leq N_{max}$$

$$2(d+1)^{h-1} - 1 \leq N \leq (2d+1)^h - 1$$

e passando ai logaritmi si ha

$$\lceil \log_{2d+1}(N+1) \rceil \leq h \leq \left\lfloor \log_{d+1}(\frac{N+1}{2}) \right\rfloor + 1$$

Esempio Supponiamo di avere le seguenti dimensioni:

- Chiave k_i: 8byte
- RID: 4byte
- entry (k+RID): 8+4=12byte
- PID: 2byte
- Pagina p: 4096byte

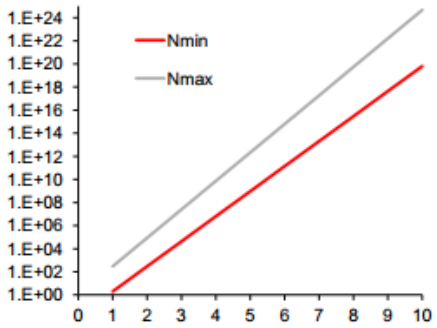
Si ottengono i seguenti risultati.

$entry \cdot 2d + PID \cdot (2d+1) = 12 \cdot 2d + 2 \cdot (2d+1) = 4096$ voglio calcolare quante entry **d** ci stanno in ogni nodo (nel tentativo di riempirlo completamente *2d*), considerando che siamo in un nodo interno, formato dunque da **entry** e **puntatori** ai nodi *figli*. Dunque 12*2d indica il numero di entry all'interno del nodo, 2*(2d+1) il numero dei puntatori ai figli (ricorda che un nodo di *m entry* ha *m+1* figli).

$d = \left\lfloor \frac{4096-2}{24+4} \right\rfloor = 146$ posso mettere fino a 146 entry in ogni nodo.
Se ho $N = 10^9$ la ricerca di un valore di chiave richiede al massimo $\left\lfloor \log_{147} \frac{10^9}{2} \right\rfloor + 1 = 5$ operazioni di I/O (una ricerca binaria richiederebbe 22 accessi supponendo di avere le pagine piene).

Una considerazione importante è la **variabilità di h (l'altezza)**. Fissati N e d (Fig. 14), la differenza tra altezza massima (quindi albero pieno a metà $m = d$) e altezza minima (albero completamente pieno $m = 2d$) è molto limitata (di ~ 1). Dunque anche se riempiamo l'albero nel modo peggiore possibile (quindi creando altezza massima) si avrebbe solo un nodo in più da visitare.

Figure 14:



P	d	N = 1000 h_{min}	N = 1000 h_{max}	N = 1000000 h_{min}	N = 1000000 h_{max}	N = 1000000000 h_{min}	N = 1000000000 h_{max}
512	18	3	3	5	5	7	8
1024	36	2	2	4	4	6	6
2048	73	2	2	4	4	5	5
4096	146	2	2	3	3	5	5
8192	292	2	2	3	3	4	4
16384	585	1	2	3	3	4	4

2.2.6 Ricerca per Intervallo

Supponiamo attraverso l'esempio in Fig. 13 di voler cercare le *entry* che hanno un valore di chiave tra 40 e 150. Da come si nota dall'immagine bisogna arrivare fino alla *foglia* che contiene 40, a quel punto bisogna fare **backtracking** e tornare in cima all'albero, alla radice, prendere la strada alternativa e scendere per fino alla foglia per trovare i valori 120 e 130, risalire ancora una volta e prendere il valore 150. Questo processo causa molta inefficienza poichè, si ricordi, che ogni volta che si scende in un nodo figlio, il nodo in memoria viene sostituito da esso: dover ritornare indietro causa il dover ricaricare nodi non voluti ma necessari per il backtracking. Ci si può accorgere che se il buffer fosse abbastanza grande da riuscire a caricare più nodi contemporanemanete invece che uno alla volta allora l'inefficienza cala, ma non è il modo migliore di operare.

La ricerca di un intervallo presuppone dunque che partendo dalla radice, si visiti l'albero *in ordine*, passando più volte attraverso gli stessi nodi a causa del backtracking. Il punto focale dell'inefficneza è data dal fatto che le **RID** non sono memorizzate solo nelle foglie (poichè se così fosse basterebbe sapere le pagine in cui i valori sono memorizzati e non sarebbe necessario il backtracking).

Si può dunque fare di meglio?

2.3 B^+-tree

In parole povere il **B^+-tree** (da ora in avanti abbreviato in B+tree) è un *b-tree* in cui le entry vengono memorizzate solo nelle foglie.

Le principali caratteristiche sono:

1. le coppie (k_i,r_i) sono tutte *contenute nelle foglie*. Questo comporta un'altezza maggiore rispetto al *b-tree* poichè ho bisogno di più foglie per poter memorizzare tutte le *entry*.

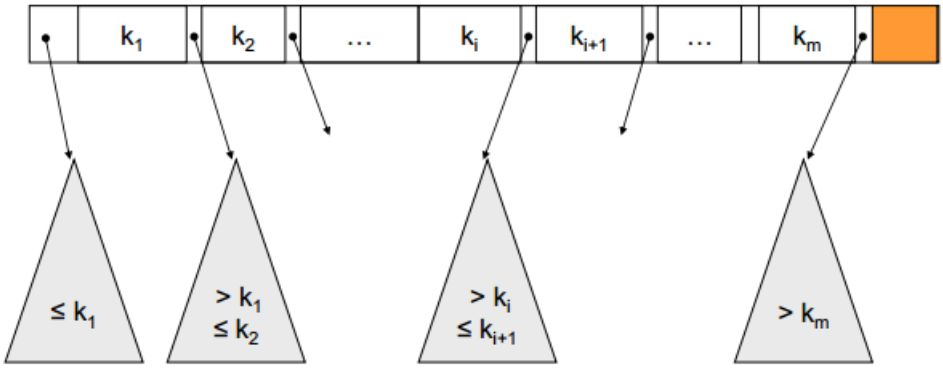
2. le foglie sono *collegate a lista* (eventualmente doppia) mediante puntatori (**PID**) per favorire la *ricerca di intervallo* (vedremo più avanti).

3. i nodi interni *contengono solamente valori delle chiavi* (non necessariamente corrispondi a dati esistenti). Questo comporta un aumento della capacità dei nodi interni (non devono più memorizzare entry, ma solo valori di chiave k) e di conseguenza diminuisce l'altezza (rispetto al *b-tree*)

Il secondo punto, oltre a favorire la ricerca di intervallo, ci fa supporre di avere, al livello delle foglie, la *lista ordinata delle entry*: è come se avessimo *paginato* un indice *single-level* e su questo avessimo creato un indice *multi-level sparse* (e in effetti ogni volta che creo un indice multi-livello sparso sto effettivamente creando un B+tree).
Il primo e il terzo punto sembrano in contraddizione, in quanto nel primo viene detto che l'albero risulta più alto, mentre nel terzo che l'albero risulta più basso. Effettivamente aver aumentato il numero delle foglie implica necessariamente avere un'altezza maggiore, ma contrapposto a questo, aver permesso ai nodi interni di contenere solo valori di chiavi (dunque solo dei k) e non più *entry* (coppie $<k_i,r_i>$) aumenta lo spazio a disposizione dei valori k nei nodi interni: ricordando che se le *entry* (in questo caso valori di chiave k) di un nodo sono m il numero di figli è m+1, allora si può capire che effettivamente l'albero si riduce in altezza poichè il suo *fan-out* è maggiore e questo porta ad avere una larghezza maggiore e dunque una riduzione dell'altezza a livello pratico.

Il formato dei nodi interni (Fig. 15) è molto simile a quello del semplice *b-tree* (Fig. 11) in cui $k_1 < k_2 < \ldots < k_m$. Le differenze riguardano ciò che viene memorizzato all'interno del nodo: non più le *entry* (k_i,r_i) ma solo i valori di chiave k_i, e il fatto che i sotto alberi contengono valori $\leq$ o $\geq$di ki (dovuto al fatto che le *entry* sono memorizzate nelle foglie, quindi i valori k_i devono essere presenti nei sotto alberi).

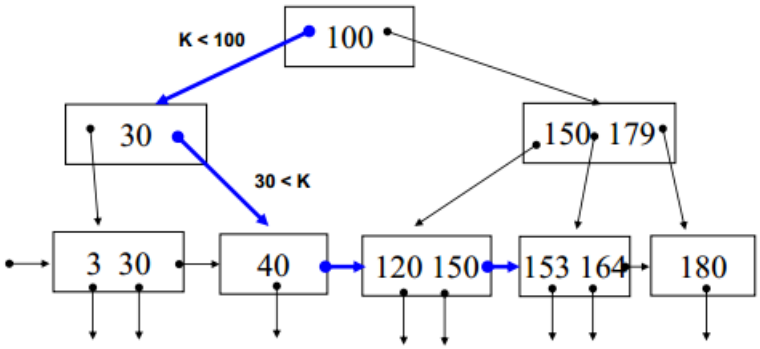
Figure 15:



2.3.1 Algoritmi di ricerca

Introduciamo il funzionamento degli algoritmi di *ricerca per chiave* e di *ricerca di intervallo*, facendo riferimento per entrambi gli algoritmi alla figura qui mostrata (Fig. 16).

Figure 16:



Notiamo prima però, come già detto, che le foglie sono collegate una all'altra attraverso liste (doppiamente) concatenate.

Ricerca di chiave Precisiamo subito che a differenza del *b-tree* in cui la ricerca di chiave aveva $costo \leq altezza\,dell'albero$, nel *B+tree* questo non è più vero, in quanto ora il costo è esattamente uguale all'altezza dell'albero $costo = altezza\,dell'albero$ poichè le *entry* si trovano nelle foglie, dunque devo sempre e comunque arrivare all'ultimo livello (delle foglie).

L'algoritmo parte dalla radice (in generale mantenuta in RAM). Si cerca la chiave k tra quelle del nodo corrente:

- Se siamo in un nodo interno sostituisci il nodo corrente con il *i-esimo* nodo figlio in cui $k_{i-1} < k \leq k_i$
- Se siamo in una foglia e il valore k è presente: *fine* (Result: **trovato**)
- Se siamo in una foglia e il valore k non è presente: *fine* (Result: **non trovato**)

Ricerca di intervallo Supponiamo di voler cercare nell'intervallo $[k_{low}, k_{high}]$. Bisogna effettuare come prima cosa una *ricerca di chiave* del valore $\mathrm{k_{low}}$ prendendo il primo valore $k \geq k_{low}$ (poichè non è detto che $\mathrm{k_{low}}$ esista). Siccome ora le foglie sono collegate a lista (doppiamente) concatenata, possiamo evitare di effettuare il **backtracking** come avveniva nel *b-tree*, poichè basterà *scandire sequenzialmente le foglie* fino a quando non troviamo il primo valore $k > k_{high}$. L'insieme di tutti i **RID** incontrati danno il risultato della *ricerca di intervallo*. L'esempio in figura (Fig. 16) mostra effettivamente una ricerca di intervallo per [30,160].

Alcune considerazioni:

- Dato che le foglie sono collegate logicamente attraverso una lista (doppiamente) concatenata, sarebbe possibile utilizzare il *b+tree* per memorizzare direttamente il *file data*, quindi al posto di inserire *entry* nelle foglie,

potremmo inserire direttamente i *record* (ordinati per un determinato attributo). Inoltre la memorizzazione ordinata così fatta crea un indice *clustered*.

- Cosa succede se l'indice è *unclustered* (dunque non è ordinato secondo l'ordine della tabella associata)? Se si ricerca per intervallo potrebbe capitare che venga restituito più volte uno stesso **PID** (ricordiamo che il **RID** è formato dalla coppia (PID, slot) quindi RID diversi potrebbero avere lo stesso PID): di conseguenza verrà richiesto al *dbms* di caricare in momenti successivi la stessa pagina per poter risolvere l'interrogazione.

- Cosa succede se l'indice è *sparse* (dunque le *entry* sono in numero minore dei *record* della tabella associata)? Un indice *sparso* è spesso e volentieri (se non sempre) anche un indice *primario* (quindi fatto su attributi UNIQUE): in una situazione di questo tipo potrebbe dunque essere conveniente memorizzare solamente **PID** e non più i **RID**, poichè una volta caricata la pagina nel buffer, leggere le informazioni ha costo che possiamo considerare 0 rispetto a una operazione di I/O.

2.3.2 Inserimento nel B+tree

Come si inserisce dunque una nuova *entry* in un *B+tree* (l'algoritmo che vedremo è scalabile, ma più complicato, a un semplice *b-tree*)? Bisogna ricordare che sono le operazioni di *inserimento* e di *cancellazione* che mantengono l'albero *perfettamente bilanciato* e *ordinato*.

Supponiamo di voler inserire una nuova *entry* (k,r). La procedura di inserimento procede innanzitutto cercando la foglia in cui inserire il nuovo valore di chiave k. Trovata la foglia si distinguono due casi:

1. Nella foglia c'è posto per la memorizzazione, cioè la foglia contiene meno di 2d entry: la nuova coppia (k,r) viene inserita e la procedura termina

2. Nella foglia non c'è più posto, cioè la foglia contiene già esattamente 2d entry (foglia in *overflow*).

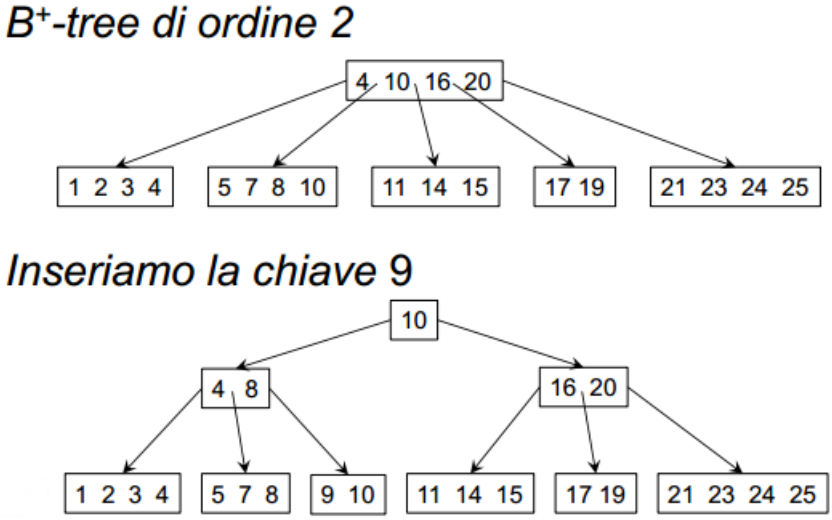
Analizziamo questo secondo caso che è ovviamente il caso di interesse.

Split di una foglia La foglia F in *overflow* viene divisa in due foglie (F_L e F_R). Ciascuna foglia conterrà circa (poichè essendo 2d pari e volendo aggiungere una nuova entry bisognerà ridistribuire in d e d+1 entry) la metà delle *entry* di F. Viene individuato il valore *mediano* k_c delle entry di F (normalmente l'indice c corrisponde all'ordine d $c = d$ - Attenzione: dire $k_c = d$ è diverso). In F_L vanno tutte le entry con $k \leq k_c$. In F_R vanno tutte le entry con chiave $k > k_c$. Nel nodo padre di F il puntatore a F viene sostituito dai due puntatori a F_L e F_R separati dal valore di k_c. Questa tecnica è chiamata **split**. Si noti che lo *split* di una foglia comporta il passaggio da una foglia completamente piena ($m = 2d$) a due foglie riempite solo a metà ($m = d$)

Cosa succede se il nodo padre di F non ha spazio per contenere questi nuovi dati?
Avviene una **propagazione dello split**, cioè il nodo padre è un nodo che va in *overflow*: si agisce come in precedenza applicando uno *split*. Ne consegue che lo *split* si propaga ricorsivamente verso l'alto. Il caso limite lo si ha quando anche la radice risulta *piena*: si divide dunque la radice in due e si crea un nuovo *nodo radice*. Effetto: l'albero **cresce in altezza**. *Osservazione:* il fatto che la radice può non rispettare il vincolo minimo di occupazione è dovuto proprio a questo effetto.

In figura (Fig. 17) troviamo un esempio di split.

Figure 17:



Costo dell'inserimento Il costo dell'inserimento è diverso in presenza o in assenza dello *split*, infatti ogni split causa 2 scritture in più. [Notazione: h=altezza dell'albero]

Senza split risulta essere: h letture (per trovare la foglia in cui inserire l'entry) + 1 scrittura (scrivere l'entry) [Senza split: h+1].

Con split:

- Nel caso peggiore si effettua la ricorsione fino alla radice: h letture (per trovare la foglia in cui inserire l'entry) + $(2h+1)$ scritture (poichè come detto ogni split provoca 2 scritture in più, bisogna risalire dalla foglia fino alla radice (distanti h) applicando lo split ad ogni livello, e infine il +1 per scrivere effettivamente l'entry) [Caso peggiore: h+2h+1]

- Per il calcolo del caso medio dobbiamo ricordare alcuni fattori: se l'albero ha **b** nodi, allora vuol dire che ci sono stati **b-1** split. Dato che il numero minimo di *entry* $N_{min} = 1 + d \cdot (b-1) \leq N$ otteniamo che il numero

medio di split è $\frac{b-1}{N}$ ovvero di circa $\frac{1}{d}$ (ho uno split ogni d *inserimenti*). Ho dunque h letture e $1 + \frac{2}{d}$ scritture. [Costo medio: $\leq h + 1 + \frac{2}{d}$]

Da come si intuisce, per avere pochi *split* conviene avere nodi grandi (così da avere un d maggiore) comportando però i soliti problemi di frammentazione interna. Inoltre è giusto chiarire che gli *split* difficilmente arrivano a propagarsi alla *radice*, mentre sono molto più comuni al livello delle foglie.

Ridistribuzione Esiste un'alternativa alla tecnica di *split*, la **ridistribuzione**. Invece di effettuare uno split vengono ridistribuite alcune entry ai nodi fratelli, cioè nodi vicini figli dello stesso padre, che non sono in *overflow.* Il tutto è permesso anche dal fatto che le foglie sono collegate tra loro attraverso lista doppiamente concatenata. Il costo della ridistribuzione rimane uguale a quello di uno *split* perchè comunque modifico 3 nodi: i due nodi foglia fratelli per la ridistribuzione delle *entry* e inoltre modifico anche il *padre* poichè è necessario modificare anche il valore mediano k_c che separava i due figli fratelli: il vantaggio è dovuto al fatto che a differenza dello *split* che può propagarsi al "nonno", la ridistribuzione si ferma con la modifica del padre. Bisogna notare che un nodo ha al massimo due fratelli (si considerano sempre e solo nodi vicini figli dello stesso padre): cosa succede però se entrambi i fratelli sono completamente pieni e quindi non possono ospitare nuove *entry*? In questo caso bisogna per forza applicare la tecnica dello **split**. In effetti l'utilizzo di entrambe le tecniche porta a buone performance e ad avere un albero mediamente "più pieno" (rispetto ad applicare solo lo *split*). Infine, tipicamente, il *padre* contiene già le informazioni sullo stato di riempimento dei figli migliorando ulteriormente le performance (cosicchè la foglia che chiede la ridistribuzione non deve andare ad indagare se uno dei fratelli ha lo spazio necessario per accettare *entry*).

2.3.3 Cancellazione nel B+tree

Supponiamo di voler cancellare una entry (k, r): precisiamo subito, è una supposizione un po' forte, in quanto, solitamente, la cancellazione avviene per ricerca di chiave k (che potrebbe avere dunque più entry associate), di conseguenza (come per altro abbiamo fatto in precedenza) supponiamo di essere in un indice *primario* (l'attributo è UNIQUE) in cui indicare la chiave k equivale ad indicare anche l'entry (k,r).

La procedura di cancellazione innanzitutto procede cenrcando la foglia in cui si trova il valore di chiave k. Trovata la foglia si cancella l'entry relativa distinguendo però due casi:

1. La foglia contiene non meno d entry: la procedura termina (rispetta il vincolo di occupazione minima).

2. La foglia contiene d-1 entry (foglia in *underflow*).

Analiziamo questo secondo caso che è ovviamente il caso di interesse.

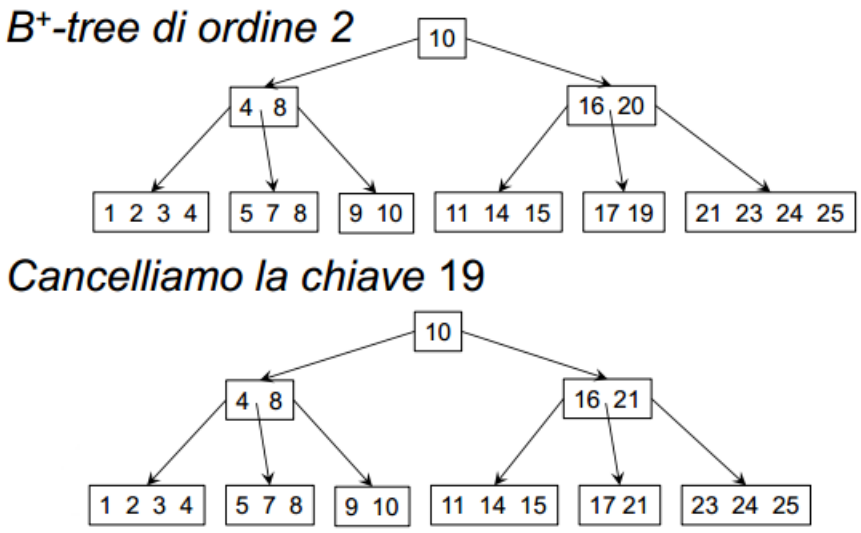
La gestione dell'*underflow* (analogamente all'*overflow* nell'inserimento) può essere gestita in due modi: **ridistribuzione** e **cancellazione**.

Ridistribuzione La ridistribuzione, come avveniva nell'inserimento, consiste nel chiedere ai fratelli (nodi vicini di uno stesso padre) almeno una *entry* per riuscire a rispettare il vincolo di occupazione minima. Questo è possibile solo se entrambi i fratelli (o l'unico fratello) ha almeno *d+1* entry, in caso contrario (*d entry*) il procedimento non è attuabile in quanto il fratello stesso entrerebbe in *overflow*. In caso di *ridistribuzione* bisogna aggiornare il valore separatore del nodo padre (il valore k_c mediano): il numero delle *entry* del padre non varia (sto modificando un valore) e il fenomeno non viene propagato verso l'alto. In caso però che entrambi i fratelli abbiano esattamente *d entry* e la ridistribuzione non sia possibile, bisogna applicare la *cancellazione.*

Cancellazione È possibile cancellare la foglia in *underflow* in caso la *ridistribuzione* non sia possibile. La cancellazione è possibile solo se la foglia/e sorella/e ha/hanno *d* o *d+1* entry. Bisogna notare che se cancello una foglia devo eliminare dal nodo padre il puntatore alla foglia stessa e ancora una volta se il nodo padre era pieno esattamente per metà, si ritroverebbe a violare il vincolo di occupazione minima. Da questo si deduce che la *cancellazione* potrebbe *propagarsi verso l'alto* allo stesso modo in cui si propagava lo *split*. Il caso limite se la *cancellazione* arriva alla radice la quale va in *underflow* (cioè la radice aveva 1 solo figlio): in questo caso viene eliminato il nodo radice e il suo unico figlio ne prende il posto. Conseguenza: l'albero **si abbassa**.

In figura (Fig. 18) troviamo un esempio di *cancellazione*

Figure 18:



Costo di Cancellazione Il costo di *cancellazione* è diverso in caso di presenza o assenza di *underflow*, infatti ogni *concatenazione* (dovute a canecllazione) causa 1 lettura e 2 scritture in più. [Notazione: h=altezza dell'albero].

In assenza di *underflow:* h letture (per trovare la foglia in cui si trova l'entry da cancellare) + 1 scrittura (cancellazione dell'entry) [Senza split: h+1].

In presenza di *underflow:*

- Nel caso peggiore: la *cancellazione* si propaga fino alla radice, avviene dunque una concatenazione per tutti i livelli tranne i primi due con una *ridistribuzione* delle *entry* nel figlio della radice, il costo massimo è di dunque $2h - 1$ letture + $h + 1$ scritture. [Caso peggiore: 2h-1+h+1]
- Per il calcolo del caso medio ho bisogno di sapere il numero medio di concatenazioni: esso ha fatto di $\frac{1}{d}$, dunque il costo medio è di $\leq h + 1 + \frac{1}{d}$ letture e $1 + 2 + \frac{2}{d}$ scritture. [Caso medio: $\leq h + 1 + \frac{1}{d} + 1 + 2 + \frac{2}{d}$]

2.3.4 Occupazione di memoria

Ogni nodo interno contiene al più *2d* valori di chiave k e dunque *2d+1* puntatori ai figli (sottoalberi). L'ordine d di un *B+tree* è quindi dato da:

$$d = \left\lfloor \frac{pagesize - PIDsize}{2 \cdot (keysize + PIDsize)} \right\rfloor$$

Al numeratore abbiamo dunque la grandezza della *pagina* alla quale sottriamo la grandezza di un *pageheader* (che in questo caso assumiamo essere fatto del solo **PID**), mentre al denominatore abbiamo due volte la grandezza del valore di chiave k e del **PID** (serve per i puntatori ai figli).

Per sapere il numero di entry che una foglia può contenere bisogna distingue il caso in cui l'indice sia *primario* e il caso in cui l'indice sia *secondario*.

Indice Primario Nel caso l'indice sia primario (e dunque non ci siano valori duplicati) in ogni foglia troviamo al più *2d* entry (k_i,r_i) e 1 o 2 puntatori ai nodi vicini (lista "doppiamente" concatenata). Ne consegue che l'ordine d delle foglie sia

$$d_{leaf} = \left\lfloor \frac{pagesize - 2 \cdot PIDsize}{2 \cdot (keysize + RIDsize)} \right\rfloor$$

Si noti che in questo caso al denominatore abbiamo il **RID** (trattandosi di un entry) e al numeratore abbiamo $2 \cdot PIDsize$ perchè supponiamo lista doppiamente concatenata.

Il numero di foglie risulta dunque essere [N=Numero di *entry*]

$$NL = \left\lceil \frac{N}{d_{leaf} \cdot u} \right\rceil$$

dove u indica il fattore di occupazione medio che normalmente vale $\ln 2$.

Conoscendo ora il numero delle foglie NL è possibile calcolare l'altezza del *B+tree* attraverso i soliti passaggi di calcolo di NL_{min} e NL_{max}

$$NL_{min} \leq NL \leq NL_{max}$$

$$2 \cdot (d+1)^{h-2} \leq NL \leq (2d+1)^{h-1}$$

e passando ai logaritmi si ottiene

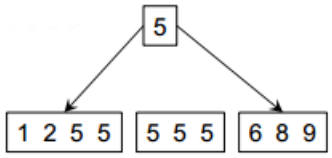
$$\lceil \log_{2d+1}(NL) \rceil + 1 \leq h \leq \left\lfloor \log_{d+1}(\frac{NL}{2}) \right\rfloor + 2$$

Indice Secondario In caso di valori duplicati occorre che ogni valore di chiave k sia associato non più ad un singolo **RID**, ma ad una *lista di RID*. Solitamente la lista è ordinata per valori di PID questo perchè, ricordando che il **RID** è formato dalla coppia <PID,slot>, non accada che in caso ci siano due *record* con uguale chiave k nella stessa pagina (identificata dal PID) per essere letti entrambi venga richiesto il caricamente della pagina due volte in momenti diversi. Cosa succede se la *lista di RID* è molto lunga? Caso limite: può capitare che una singola entry abbia dimensioni maggiori di una singola pagina. Possibili soluzioni:

- Pagine di *overflow* per la foglia in questione
- *Duplicazione* delle chiavi nell'indice
- Uso di *PID* al posto dei *RID*
- Posting file

Duplicazione delle chiavi nell'indice Invece di mantenere una lista di RID associate a una chiave k, mantengo ogni entry (k,r) separate, duplicando dunque le entry aventi la stessa chiave k. Questa soluzione richiede di modificare leggermente l'algoritmo di ricerca: bisogna dunque cercare il *primo* valore uguale alla chiave di ricerca k, quindi proseguire con la lista delle foglie (ottenendo tutti i RID). Inoltre non tutte le foglie sono "indirizzate", si noti la figura (Fig. 19).

Figure 19:



Inoltre la soluzione diventa inefficiente in *cancellazione* in quanto per eliminare una *entry* bisogna specificare anche il **RID**, e non solo la chiave di ricerca k (se utilizzassimo sempre la coppia <k,RID> per qualsiasi operazione l'indice diventa *primario*).

Utilizzo dei PID Si potrebbe pensare di mantenere un elenco di PID invece che l'elenco dei RID, includendo solo le pagine che contengono almeno un record della chiave k. Il primo vantaggio è dovuto al fatto che $PIDsize \leq RIDsize$, un secondo riguarda il fatto che il numero di PID è solitamente molto minore dei numero dei RID (dunque sono proprio di meno e la lista non potrà essere lunga), infine se l'indice risulta *clustered* (ordinato per il valore di chiave k) l'efficienza aumenta di molto sopratutto se in una singola pagina ci sono molti record con lo stesso valore di chiave k.

Posting File L'elenco dei RID viene mantenuto in un file separato (il *posting file*): ogni entry del *posting file* ha la forma (k,l) dove:

- **k** è il valore di chiave
- **l** è la lista di RID aventi **k** come valore di chiave

Le entry del *B+tree* conterranno ora riferimenti alle *entry* del *posting file*. Lo svantaggio maggiore è (oltre all'occupazione in memoria) è che si introduce un ulteriore livello di indirezione, facendo aumentare di 1 il costo di ogni singola operazione.

2.3.5 B+tree nella pratica

Finora abbiamo parlato di un *B+tree* teorico (cioè applicato a un indice *primary* e magari anche *clustered*). Ma nella pratica?

In pratica i *B+tree* devono tener conto di molte altre esigenze:

- *B+tree* come organizzazione principale dei dati
- Chiavi a lunghezza variabile
- Compressione delle chiavi
- *B+tree* multi-attributo
- Bulk-loading
- Implementazione *B+tree:* ***GiST***

B+tree come organizzazione principale Inizialmente il B+tree nacque proprio con questo scopo, cioè di introdurre i *record dati* direttamente nelle foglie dell'albero. Evidentemente questo permetteva di ordinare il *file dati* automaticamente grazie alle operazione di inserimento/cancellazione: di contro però, ad ogni operazione, i record si spostavano, e dunque il **RID** cambiava. Cambiare il **RID** implica dover aggiornare tutti gli indici che contengono la tupla associata: questo è stato il motivo principale per cui il *B+tree* venne poi solo usato per memorizzare indici.

Chiavi a lunghezza variabile Tutto ciò che abbiamo visto fino ad ora aveva come assunzione che le *entry* fossero di lunghezza fissa. Questo nella pratica è difficilmente vero:

- Chiavi di lunghezza variabile (es. *varchar*)
- Indice secondario (dunque *lista di RID* associata ad ogni entry)
- Indice come organizzazione principale dei dati (dunque ogni entry in realtà è un record quasi sempre di lunghezza variabile)

In questi casi il concetto di ordine d dell'albero perde di validità, ma questo non inficia la *ratio* utilizzata dietro il meccanismo dei *B+tree*: invece di applicare il vincolo di occupazione minima alle *entry*, esso viene applicato all'utilizzazione dello spazio (fisico) all'interno del nodo.

Compressione delle chiavi È evidente che, per minimizzare i tempi di accesso all'albero, conviene avere valori elevati di d (ordine dell'albero). Si può pensare di ridurre la lunghezza dei valori della chiave di ricerca k **nei nodi** (nelle foglie è bene non attuarlo poichè nelle foglie vengono memorizzate effettivamente le *entry*) attraverso una *compressione*, con lo scopo di aumentare il numero dei valori k inseribili all'interno di ogni nodo

Nei *B+tree non è necessario* che i nodi interni contengano valori di chiavi esistenti nei dati, essi servono prettamente a differenziare il contenuto di nodi figli adiacenti.

Se ad esempio nella foglia A abbiamo le due entry (Semenzara, RID_1) (Serbelloni Mazzanti Vien Dal Mare, RID_2) e nella foglia B abbiamo l'entry (Silvani, RID_3) si nota che basterebbe salvare nel nodo padre il valore di ricerca "Ser" per riuscire a distinguere in quale foglia andare a cercare effettivamente l'entry.

Ricerche multi-attributo Supponiamo di avere la seguente query:

```
SELECT * FROM persone
WHERE cognome=''Rossi''
AND anno>1990
```

Come possiamo utilizzare un indice per risolvere efficentemente la query? Si noti che il WHERE richiede due chiavi di ricerca, una per il cognome e una per l'anno.

Una prima soluzione che può venire in mente è quella di usare un solo indice: si recuperano attraverso l'indice i record che soddisfano il primo (o il secondo) predicato e si scelgono solo quelli che soddisfano anche l'altro. È giusto sottolineare che la soluzione può migliorare se l'indice selezionato "taglia" il più possibile: nell'esempio è più facile che siano presenti meno record il cui cognome sia Rossi piuttosto il cui anno di nascita sia superiore a 1990, ecco che la strategia migliore è di applicare un indice al campo cognome.

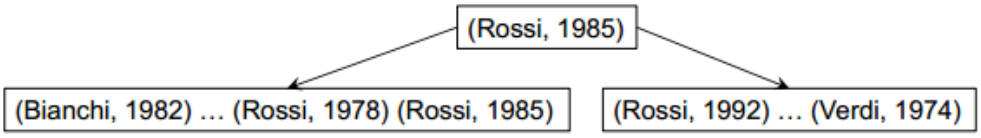
Una seconda soluzione prevede di usare invece entrambi gli indici: si recuperano separatamente i **RID** che soddisfano i due predicati (uno da un primo

indice e l'altro dal secondo) e se ne effettua l'intersezione. La soluzione diventa più efficiente se i RID sono mantenuti ordinati.

In entrambi i casi il lavoro aggiuntivo può vanificare il vantaggio di usare l'indice. Una buona soluzione è dunque quella di costruire un indice *multi-attributo*. La chiave di ricerca k è composta ora dalla concatenazione degli attributi coinvolti e l'ordinamento risulta essere quello lessicografico. **NB** solo a parità di un attributo si considera l'attributo successivo.

In figura (Fig. 20) un esempio di indice *multi-attributo* su attributi (cognome,anno).

Figure 20:



Questo indice è in grado di risolvere efficientemente interrogazioni su *cognome* e su (*cognome*,*anno*). Risolve efficientemente anche interrogazioni di tipo intervallo però solo applicate ai cognomi (Le persone che hanno cognome da 'Bianchi' a 'Rossi') o ai cognomi e anno (I 'Rossi' nati dal '1985' al '1990'). L'indice in figura non è però in grado di risolvere efficientemente interrogazioni solo su *anno*: il motivo è ovvio, l'ordine degli attributi è importante per permettere l'ordinamento lessicografico. In generale se ho n attributi posso costruire n! indici ed è compito dell'*amministratore* del database scegliere gli indici opportuni.

Bulk-Loading Spesso si ha la necessità di costruire (o ricostruire) un indice ex novo: infatti molto spesso la decisione di costruire un indice non avviene in fase di creazione del database ma solo in inseguito (ad esempio ci accorgiamo ci accorgiamo che una query è lenta) o ancora quando richiediamo l'ordinamento fisico di un file con modifiche ai relativi RID. Inserire una per una le *entry*, come visto, provoca *mediamente* uno split ogni **d** record inseriti (poichè la probabilità di split ricordiamo è $\frac{1}{d}$). Evidentemente creare un indice effettuando l'inserimento uno a uno non risulta essere la soluzione migliore. Ecco dunque che si può ricorrere alla tecnica del **bulk-loading**: si crea una lista di coppie (k_i,RID_i) e la si ordina per valori della chiave (**N.B.** non ci stiamo preoccupando al momento di capire come ordinare una lista di coppie che non sono in memoria centrale, facciamo l'assunzione che sia possibile farlo, più avanti vedremo come si risolve il problema), questa lista (opportunamente paginata) corrisponde al livello delle foglie del nostro B+tree (N.B. è possibile non usare le foglie direttamente al 100% ma lasciare spazio libero così da non provocare un immediato split al prossimo inserimento). A partire ora dal livello delle foglie viene creata una nuova lista (sostanzialmente i separatori) di coppie (k_i,PID_i): paginando

opportunamente anche questa lista abbiamo costruito il livello superiore alle foglie. Il procedimento prosegue via via fino ad ottenere la radice. Osserviamo dunque che stiamo costruendo l'indice dal basso verso l'alto.

2.4 GiST

Il **GiST** (**G**eneral**i**zed **S**earch **T**ree), creato da *Hellerstein*, *Naughton*, *Pfeffer* nel 1995 (come si vede è successivo al B-tree e B+tree) non è una implementazione del *B+tree* ma un *framework* del tuto generale che permette di creare alberi; non è dunque uno specifico metodo di accesso bensì una struttura generalizzata che a seconda di come viene istanziata può comportarsi come un *B+tree* o un *R-tree* (vedremo più avanti) ecc...

Il GiST permette dunque di capire i principi costruttivi degli alberi con l'obiettivo principale di semplificare lo sviluppo di diversi metodi di accesso (e non dunque di definire un nuovo tipo di indice).

Esempio: nel sistema *postgres* (dbms) i *B+tree* vengono implementati in circa 3000 righe di codice C. Lo stesso *B+tree* implementato come istanza del GiST richiede circa in totale 500 righe di codice (questo perchè molta parte implementativa viene gestitata da GiST stesso).

2.4.1 Concetti alla base di GiST

Invece di considerare specifiche *query,* GiST ne generalizza il concetto portandolo a livello di *predicato* (**p**). Ogni **nodo** del GiST contiene una serie di *entry* formate dalla coppia **(p,ptr)** dove

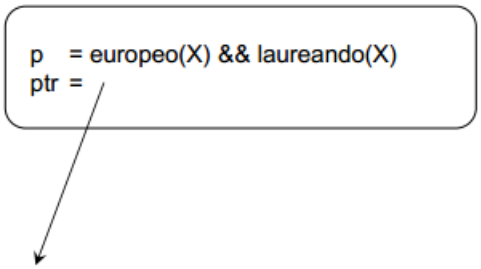
- **p** è un *predicato generico* (una chiave poi nel nostro B+tree)
- **ptr** è *puntatore generico*: può dunque puntare sia a un altro nodo dell'albero (dunque a una pagina) sia direttamente a una tupla (quindi a un RID)

L'accesso a un sottoalbero individuato dal puntatore associato a una chiave avviene **solo se** la chiave è *consistente* con il *predicato q*, ovvero solo se la chiave non esclude la possibilità che il sottoalbero relativo possa contenere dati che soddisfino q: chiedersi se il predicato P è consistente al predicato Q vuol dire chiedersi se P può fare in modo che Q sia vero. Il **solo se** prima evidenziato ha il dovere di permettere l'accesso all'albero anche nei casi in cui il predicato P non soddisfi poi in realtà il predicato Q. Tentiamo di spiegarci meglio con un esempio sul *B+tree*: la radice contiene il valore 50 e ci chiediamo se il valore 20 esiste nell'albero (notiamo che qui il nostro predicato P riguarda l'esistenza del 20); sicuramente essendo $20 < 50$, sono sicuro che devo accedere al sottoalbero sinistro e sicuramente non al destro perchè anche se non sono sicuro che il 20 esista davvero nell'albero poichè lo scoprirò solo al livello delle foglie (e qui si spiega il **solo se**), esso (il 20) è *consistente* con il sottoalbero sinistro ma non con il sottoalbero destro.

Proprietà fondamentale di un GiST è la **monotonicità del predicato p:** cioè se il predicato p vale per un determinato nodo (che sia foglia o nodo interno)

esso dovrà valere **per forza** per tutti i suoi antenati (propri). Se così non fosse mi potrei ritrovare che il predicato p risulta essere consistente per il nodo N ma non per il padre la qual cosa mi porterebbe a scartare l'accesso al nodo padre stesso facendomi perdere informazioni poichè p consistente con il nodo N. I predicati che troviamo a livelli delle foglie sono più *restrittivi* (specifici) di quelli che troviamo nei livelli superiori che risultano più *laschi*: questo implica che è difficile "buttare" sottoalberi a livelli alti, e man mano che si scende l'albero fino al livello foglie, la probabilità aumenta fino a diventare certezza.

Figure 21:



Vediamo un esempio (Fig. 21). Nella figura abbiamo rappresentato un nodo contentente l'entry (p,ptr) specificate. Ci chiediamo ora se alcuni predicati (query) Q sono consistenti con il predicato p.

- Q=italiano(X) && studente(X) è consistente con p poichè italiano è anche un europeo e studente potrebbe essere un laureando (il condizionale è d'obbligo, solo scendendo poi l'albero lo scopriremo)
- Q=messicano(X) && matricola(X) non è consistente con p poichè messicano non è europeo
- Q=francese(X) && lavoratore(X) è consistente con p di nuovo perchè francese è europeo e un lavoratore potrebbe essere anche un laureando
- Q=messicano(X) OR studente(X) è consistene con p poichè anche se il messicano non è europeo, uno studente potrebbe benissimo essere un laureando.

Cosa potrebbe ora contenere il figlio di questo nodo (cioè il sottoalbero puntato da ptr)?

Il nodo figlio potrebbe avere entry con predicato p=italiano(X) && laureando(X) e per monotonicità del predicato ciò che risulterebbe vero per il figlio lo risulterebbe anche il padre. Il nodo figlio invece non può contenere entry con predicato p=italiano(X) && studente(X) poichè non tutti gli studenti sono laureandi (**n.b.** qui ci stiamo chiedendo quale entry può avere il figlio non se il predicato è consistente con il nodo mostrato in figura - **n.b.2** come detto prima,

man mano che si scende nell'albero i predicati diventano più stringenti ecco perchè italiano(X) && studente(X) non può essere figlio del nodo mostrato, poichè studente è una generalizzazione di laureando) e non può contenere neppure un entry con predicato p=italiano(X) poichè non tutti gli italiani sono laureandi.

2.4.2 Proprietà di un GiST

Indipendentemente dalla specifica istanziazione, ogni Gist gode delle seguenti proprietà:

- Un GiST è un *albero paginato perfettamente bilanciato*
- Una *entry* di un *nodo intermedio* è una coppia (p,ptr) con
  - **p** predicato usato come chiave di ricerca
  - **ptr** puntatore a un altro nodo del GiST
- Una *entry* di un *nodo foglia* è una coppia (p,ptr) con
  - **p** valore di chiave
  - **ptr** puntatore a una tupla (oggetto del database) che soddisfa p
- Ogni nodo, ad eccezione della radice, contiene un massimo di **M** entry e un minimo di $f \cdot M$ entry, con $\frac{2}{M} \leq f \leq \frac{1}{2}$ (f=fattore di caricamento). Come si nota a differenza del B+tree ora il fattore minimo di riempimento può essere anche minore del 50%. Si nota che il massimo valore che f può assumere è proprio del 50% (cioè il massimo sul vincolo di riempimento minimo è $\frac{1}{2}$) mentre invece il minimo è 2 (posso mettere come minimo al vincolo minimo di riempimento 2 entry). Come accadeva nel B+tree in caso di lunghezza variabile si usano fattori non legati al numero di entry ma alla loro dimensione e al loro valore di riempimento.
- La *radice*, se non è una foglia, ha almeno due *entry*
- Per ogni entry (p,ptr) di un nodo intermedio, p vale per pgni tupla raggiungibile via ptr: cioè se ho il mio nodo il cui figlio è identificato dal determinato ptr associato, il predicato p vale anche per ogni entry del nodo figlio prima indicato.

2.4.3 Realizzazione concreta

Alla base del GiST vi è la definizione di una serie di metodi relativi alla gestione:

- dei valori di chiave (**key methods**)
- dell'albero (**Tree methods**)

La definizione del GiST specifica solo i *Tree methods*, la specifica dei *Key methods* è eseguita quando si istanzia il GiST per gestire uno specifico tipo di chiavi (Esempio: tipo di chiavi a valori interi istanzierà un *B+tree*, tipo di chiavi a intervalli multi-dimensionali istanzierà un *R-tree*).
Poichè i *Key methods* sono invocati da *Tree methods* è necessario standarizzare l'interfaccia dei primi.

2.4.4 Key Methods

Consistent (Utilizzato in Ricerca) `Consistent(E,q)`

- **Input:** Entry E = (p,ptr) - q: predicato di ricerca q
- **Output:** `IF (p&q==false) THEN false ELSE true`

Lo scopo di *Consistent* è eseguire un "*pruning*" (potatura) dello spazio di ricerca (ovvero eliminare sotto-alberi). Se il predicato di un sotto-albero non è consistente con la query (predicato), si evita di accedere all'intero sotto albero. Nel caso il test `(p&q==false)` risulti oneroso è comunque possibile lavorare con approssimazioni conservative ovvero rispondere *true* anche se poi risulterà essere *false*, questo poichè se avessimo risposto *false* avremmo potuto tagliare sotto-alberi che contenevano invece informazioni necessarie, avendo invece risposto *true* ci addentriamo nel sottoalbero e, in caso, solo al livello delle foglie ci accorgeremo se effettivamente risulti essere true o false (serve insomma a mantenere la monotonicità della consistenza sui predicati). Ciò incide a livello di prestazioni ma non di correttezza poichè si accede a un sottoalbero anche se i suoi dati non contribuiscono al risultato della query. *Consistent* (così come gli altri metodi) è specificato per lavorare con predicati di complessità arbitraria anche se poi in pratica i predicati possono essere *ristretti* per migliorarne l'efficienza.

Consistent applicato al B+tree

Ogni *predicato p* delle *entry* è in realtà un intervallo **[x,y[**. Se la query q è un valore **v** *Consistent* restituisce *true* se e solo se $x \leq v < y$ (**NB** se la *entry* invece di aver salvato un intervallo, avesse salvato un singolo valore **x** si potrebbe ancora applicare la specifica prima descritta utilizzando l'intervallo **[x,x[**). Se la query q è invece un intervallo **[v,w[** *Consistent* restituisce *true* se e solo se $x < w$ o $y > v$ (se l'interesezione tra i due intervalli è non vuota).

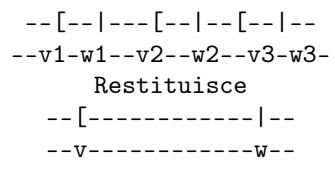
Union (Utilizzato nella creazione di predicati) `Union(P)`

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

Scopo di *Union* è di fornire l'informazione necessaria a caratterizzare il predicato di un nodo padre a partire dai predicati dei nodi figli. In generale il *predicato r* può essere logicamente derivato come un predicato per cui vale $(p_1 | \ldots | p_n) \Rightarrow r$.

Union applicato al B+tree

Dato l'insieme di entry $P = \{([v_1, w_1[, ptr_1), \ldots, ([v_n, w_n[, ptr_n)\}$ restituisce un nuovo *predicato* r sotto forma di un intervallo $r = [min\{v_1, \ldots, v_n\}, max\{w_1, \ldots, w_n\}[$.



NB L'*Union* non sta restituendo la vera unione di tutti gli intervalli, ma solo l'intervallo creato dal minimo e dal massimo (rende più generale il predicato dei figli).

Compress (Utilizzato nella compressione delle chiavi) `Compress(E)`

- **Input:** Entry E = (p,ptr)
- **Output:** una nuova Entry E'=(p',ptr) con p' rappresentazione compressa di p

Data una *entry* ne restituisce una nuova che abbia lo stesso puntatore *ptr* ma un diverso predicato *p'* rappresentazione compressa del predicato originale *p*. Scopo di *Compress* è fornire una rappresentazione più efficiente del *predicato p*. Esempio: separatori al posto di intervalli totalmente ordinati (invece che indicare l'intervallo [x,y[posso indicare solo x) o prefissi a partire da stringhe ("Serbelloni Mazzanti Vien Dal Mare" viene trasformato in solo "Ser"). La compressione diminuisce l'informazione rappresentata e può essere sia con perdita che senza perdita.

Decompress (Utilizzato nella decompressione delle chiavi) `Decompress(E)`

- **Input:** Entry E' = (p', ptr) con p' derivato da una compressione $p' = Compress(p)$
- **Output:** una nuova Entry E = (r, ptr) con $p \Rightarrow r$

La compressione in generale non è *lossless* (cioè in generale è con perdita),questo richiede che il *predicato r* della nuova *entry* sia una derivazione logica dal *predicato p originale* $p \Rightarrow r$ cioè se data una query Q, Q non è consistente con p allora Q non è consistente con r. Il caso più semplice di *decompressione* è che questa risulti essere la funzione identità.

Penalty (Utilizzato nell'inserimento) Penalty(E_1,E_2)

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

Data una nuova *entry* (o un nuovo nodo) devo scegliere in quale sottoalbero (o nodo) inserirla, allora *Penalty* restituisce una valore intero di "penalità" che verrà confrontato con diversi altri valori di penalità per altri nodi permettendo di stabilire la scelta corretta.

Penalty applicato al B+tree

Nel B+tree *Penalty* descrive un valore associato a quanto dobbiamo ingrandire l'intervallo del padre per poter includere l'intervallo dell'entry che stiamo inserendo.

Date due Entry: $E_1 = ([x_1, y_1[, ptr_1)$ e $E_2 = ([x_2, y_2[, ptr_2)$

Penalty restituisce un valore che indica di quanto dobbiamo ingrandire il primo intervallo per includere il secondo (normalmente si può ingrandire sia a destra che a sinistra).

- Caso generale: *Penalty* restituisce $max\{y_2 - y_1, 0\} + max\{x_1 - x_2, 0\}$
- E_1 è la prima *entry* del suo nodo: *Penalty* restituisce $max\{y_2 - y_1, 0\}$
- E_1 è l'ultima entry del suo nodo: *Penalty* restituisce $max\{x_1 - x_2, 0\}$

PickSplit (Utilizzato per lo split) `PickSplit(P)`

- **Input:** Insieme di M+1 entry
- **Output:** Due insiemi di *entry* P_1 e P_2 di cardinalità $\geq f \cdot M$

Ho un insieme di M+1 entry poichè un nodo è andato in *overflow* e devo dunque dividirlo in due, esso restitusce un insieme di due entry con cardinalità maggiore del fattore minimo di occupazione che avevo imposto $\geq f \cdot M$. *PickSplit* implementa la vera e propria strategia di split che non viene specificata a questo livello e anche in questo caso, tipicamente, esso tenta di minimizzare similmente a come fa *Penalty*

PickSplit applicato al B+tree

Nel B+tree i due nuovi insieme di entry P_1 e P_2 si divideranno le entry in modo tale che P_1 contenga le prime $\frac{M+1}{2}$ entry e che P_2 contenga le rimanenti. Nel caso di *entry* di lunghezza variabile invece di utilizzare il numero di *entry* si utilizzano criteri legati alla loro dimensione, questo però potrebbe portare a violare il vincolo di utilizzazione minima. (Ad esempio: ho una *entry* che deve essere messa o in P_1 o in P_2. Se messa in P_1 essa può essere compressa però portando in *overflow* P_2 perchè non abbastanza riempito, mentre se messa in P_2 essa viene inserita integra portando però in *overflow* P_1).

2.4.5 Tree Methods

I *Tree methods* si richiamano fra loro e utilizzano alcuni dei *key methods* prima definiti. Si assume implcitamente che le chiavi vengano compresse in fase di scrittura e decompresse in fase di lettura. I *Tree methods* in breve sono:

Search permette la ricerca e utilizza *Consistent*

Insert permette l'inserimento e utilizza **ChooseSubtree**, **Split** e **AdjustKeys**

ChooseSubtree permette di determinare il nodo in cui inserire e utilizza *Penalty*

Split permette lo split e utilizza *PickSplit* e *Union*

AdjustKeys permette di sistemare le chiavi di ricerca e utilizza *Union*

Delete permette di eliminare una entry e utilizza **Search** e **CondenseTree**

CondenseTree permette la "ridistribuzione" e l'abbassamento dell'altezza dell'albero in caso di cancellazione e usa **AdjustKeys** e **Insert**

Search `Search(R,q)`

- **Input**: (sotto-)albero con *radice* R e *predicato* (query) q
- **Output**: Tutti i record (entry) che soddisfano (sono consistenti) con q

```
if R non è una foglia
-for each E in R
--if Consistent(E,q) then Search(*(E.ptr),q)
else for each E in R
-if Consistent(E,q) then
--aggiungi *(E.ptr) al risultato
```

Algoritmo a parole:
Se R non è una foglia per ogni *entry* in R: se l'*entry* è consistente con il predicato faccio il *Search* sul relativo figlio
Se R è una foglia, per ogni *entry* in R: se è l'*entry* è consistente con il predicato aggiungo il puntatore (che sarà un RID) al risultato.

Che succede però se sto cercando un intervallo? Sembrerebbe essere tornato in causa il **backtracking** tanto agoniato. In realtà per domini lineari (totalmente ordinati) come nel caso dei B+tree, la specifica del GiST pervede un'estensione del **Search** più efficinete che sfrutta la contiguità delle foglie per risolvere la ricerca per intervallo. In particolare la **Search** raggiunge la prima foglia *consistente* con la query q, dopodiché sfrutta il collegamento a lista delle foglie fino a che non si raggiunge la prima foglia *non consistente* con il predicato q.

Insert `Insert(R,E,l)`

- **Input**: Albero con *radice* R, la Entry E da inserire, il livello l in cui inserirla
- **Output:** un nuovo albero con la Entry E inserita al livello l.

```
N = ChooseSubtree(R,E,I)
if E può essere inserita in N then inserisci E in N
```

```
else Split(R,N,E)
AdjustKeys(R,N)
```

Algoritmo a parole:

Scelgo il nodo N a livello l in cui inserire la mia entry E attraverso il metodo *ChooseSubtree*
Se E può essere inserita in N allora inserisci
Altrimenti esegui lo *Split*(R,N,E)
Infine aggiusta le chiavi attraverso *AdjustKeys*

L'algoritmo di inserimento viene usato sia per inserire nuove entry, sia per re-inserire *entry* "orfane" risultanti da *underflow*. Per tale motivo viene passato in input anche il livello dell'albero in cui inserire l'*entry* (poichè le entry orfane potrebbero trovarsi a livelli superiori rispetto alle foglie) con la convezione che le foglie siano al livello 0. In caso di *overflow* viene attivata la procedura di *Split* (che può propagarsi) portando i cambiamenti anche verso l'alto.

ChooseSubtree `ChooseSubtree(R,E,I)`

- **Input**: Albero con radice R, l'entry E da inserire, il livello l in cui inserirla
- **Output**: nodo N al livello l in cui inserire E.

```
if R è al livello l return R
else scegli tra tutte le entry F=(p',ptr') in R quella per cui Penalty(F,E)
è minimo
return ChooseSubtree(*(F.ptr'),E,I)
```

Algoritmo a parole:

Se R (la radice dell'albero/sottoalbero) si trova al livello l ritorno R (ho trovato il nodo in cui inserire E)
Altrimenti scegli tra tutte le entry F che stanno nel nodo R quella che ha *Penalty* minima con la entry E da inserire e richiama *ChooseSubtree* sul nodo figlio associato alla F selezionata.

Il *ChooseSubtree* non torna mai indietro su una decisione ad un certo livello (fa una scelta greedy) utilizzando *Penalty* per determinare ricorsivamente il sotto-albero in cui inserire E.

Split `Split(R,N,E)`

- **Input**: Albero con radice R, il nodo N che si deve suddividere e la entry E che si deve inserire
- **Output:** Nuovo albero con N suddiviso ed E inserita.

```
P1,P2 = PickSplit({entry in N} U {E})
inserisci P1 in N e P2 in un nuovo nodo N'
p'=Union(P2)
ptr'=&N'
E'=(p',ptr')
```

```
if E' può essere inserita in Parent(N) then inserisci E' in Parent(N)
else Split(R,Parent(N),E')
F = entry in Parent(N) con F.ptr=&N
F.p=Union(P1)
```

Algortimo a parole:

Prima cosa chiamo *PickSplit* che mi restituisce i due insiemi di *entry* P_1 e P_2
Inserisci P_1 in N e P_2 in un nuovo nodo N'
Costruisco la nuova Entry E' (p',ptr') che contiene informazioni sul nuovo nodo N' che abbia p'=*Union*(P_2) e ptr'=&N'
Il padre di N ha spazio per inserire la nuova entry E'?
Se sì: inserisco E' nel padre di N
Se no: eseguo lo *Split* sul padre di N e con la nuova Entry E' (da inserire)
Infine cerco l'entry F nel padre di N che puntava al vecchio nodo ormai suddiviso e gli aggiorno il predicato p a Union(P_1)

Split usa *PickSplit* per dividere le entry di un nodo in overflow. Diamo per ipotesi che il nodo padre del nodo in overflow si trovi sullo stack quindi raggiungibile.

AdjustKeys `AdjustKeys(R,N)`

- **Input**: Albero con radice R e il nodo N dal quale si vogliono sistemare le chiavi
- **Output:** Nuovo albero con gli antenati di N con valori di chiave (predicati) corretti e accurati

```
E = entry (p,ptr) il cui ptr=&N
if N=R or E.p=Union({entry di N}) return
else E.p = Union({entry di N})
AdjustKeys(R,Parent(N))
```

Algoritmo a parole:

Trovo l'entry *E* tale per cui il suo campo ptr punti al nodo *N* di input.
Se N è la radice oppure E.p è già uguale all'*Union* delle entry di N (cioè ha il predicato già accurato) allora **return** (fine)
Altrimenti modifico il valore E.p ponendolo uguale all'*Union* delle entry di N
Infine richiamo *AdjustKeys* al padre del nodo N.

AdjustKeys ricalcola i valori di chiave (predicati) a seguito di modifiche. L'algoritmo risale ricorsivamente l'albero e termina quando si raggiunge la radice o quando trova un valore di chiave già accurato.

Delete `Delete(R,E)`

- **Input:** Albero con radice R e *entry* E = (p,ptr) da eliminare
- **Output:** Albero con l'entry E rimossa

```
Search(R,E.p)
if E non trovata return
L = nodo che contiene E
Rimuovi E da L
CondenseTree(R,L)
if R ha una sola entry
-rimuovi R
-rendi il figlio di R la nuova radice del GiST
```

Algoritmo a parole:
Cerco E.p
Se non trovo una corrispondenza allora termino (nulla da cancellare)
Identifico L come il nodo che contiene l'entry E e rimuovo l'entry dal Nodo.
Applico il *CondenseTree*(R,L) al nodo L
Se R (radice) ha una sola entry allora elimino la radice e rendo il suo unico figlio la nuova radice del GiST.

Delete mantiene l'albero bilanciato e ne riduce l'altezza se la radice, al termine di *CondenseTree* ha un solo figlio. Ovviamente abbiamo supposto di voler cancellare una determinata *entry* in presenza di predicati *univoci* (se così non fosse devo mettere nella ricerca la coppia (p,ptr) a causa di chiavi duplicate).

CondenseTree CondenseTree(R,L)

- **Input:** Albero con radice R e la foglia L da cui partire
- **Output:** un nuovo albero (che non per forza dovrà essere un GiST poichè la radice potrebbe avere un solo figlio cosa non permessa nei GiST)

```
N=L
Q={ }
if N = R goto end
else
-P = Parent(N)
-E = entry in P tale che E.ptr=&N
if #{entry di N} < f*M
-Q = Q U {entry di N}
-rimuovi E da P
-AdjustKeys(R,P)
if E non è stata rimossa da P
-AdjustKeys(R,P)
else
-N = P
-restart
for each E in Q
-Insert(R,E,level(E))
```

Algoritmo a parole:

Sia N = al nodo di cui ci stiamo occupando
Sia Q = l'insieme inizialmente vuoto delle entry orfane
Se N = R (radice) allora ho finito (cioè la radice è una foglia)
Altrimenti
Prendo il padre P del nodo N di cui ci stiamo occupando
Prendo l'entry E nel padre che corrisponde al nodo N
Se il numero di Entry nel nodo risulta minore di *f*M* (fattore minimo di occupazione), cioè il nodo N è in *underflow* allora: aggiorno la lista delle entry orfane aggiungendo quelle di N, rimuovo l'entry E dal padre P (poichè il nodo non è più presente) e aggiusto il predicato del padre P attraverso *AdjustKeys*
Se l'entry E non è stata rimossa dal padre P (cioè il nodo N non era in *underflow*) allora devo aggiornare il predicato del padre P (poichè non è stato fatto al punto precedente) attravers *AdjustKeys*
Altrimenti (cioè ho fatto la cancellazione) impongo il nuovo nodo di interesse N uguale al padre P e riparti da capo (in poche parole devo risalire la catena e controllare che anche il padre P non sia andato in *undeflow*).
Infine: inserisco tutte le entry orfane E che stavano nella lista Q nell'albero attraverso *Insert* specificando per ogni *entry* il proprio livello.

Il *CondenseTree* gestisce il reinserimento al livello originario delle entry orfane di nodi in underflow che sono state mantenute in un insieme Q a parte.

2.4.6 Prestazioni

Fino ad ora abbiamo calcolato le prestazioni della ricerca per un B+tree usato come indice primario, ma questo non avviene sempre, dobbiamo dunque capire le prestazioni in caso di un indice *secondario* (e dunque con chiavi ripetute) o in caso di ricerca per intervallo (che può essere paragonato alla ricerca su indice secondario, in quanto una volta arrivato alle foglie devo poi scorrerle per trovare i risultati all'interno dell'intervallo). In questo caso bisogna capire

- Quante siano le foglie contenenti *entry* del risultato (quindi il numero di foglie)
- Quante siano le pagine dati contenenti i record associati alle entry risultanti (quindi il numero di pagine)

Supporremo che

- Le liste dei riferimenti nelle foglie siano ordinate (per PID) così da non accedere più di una volta alla stessa pagina dati
- I valori degli attributi siano distribuiti uniformemente nel file dati, cosicché ogni valore sia ripetuto in media $\frac{N_t}{N_k}$ volte (con N_t numero di tuple e N_k numero di valore di chiavi distinti - Da adesso in poi indicheremo con N il numero di tuple a meno che non espressamente specificata la differenza). **NB** in caso i valori non siano distribuiti in maniera uniforme si può pensare che determinati *cataloghi* specializzati proprio in questo, riportino un istogramma statistico sulla ripetizione dei vari valori.

- I record siano distribuiti uniformemente nelle pagine del file dati

Stima del numero di pagine Andiamo ora a stimare il numero di pagine medio da caricare per riuscire a risolvere la nostra query (cioè il numero di pagine in cui trovo i *risultati*, ad esempio se la ricerca risponde che la query è risolta da un numero R di tuple, in quante pagine queste tuple sono distribuite? Sappiamo per certo essere un numero $\leq R$).

Se **R** sono i record da reperire in **P** pagine:

1. $\frac{1}{P}$ = probabilità che uno (specifico) dei R record si trovi in una data pagina
2. $1 - \frac{1}{P}$ = probabilità la data pagina non contenga lo specifico record dei R record
3. $(1 - \frac{1}{P})^R$ = probabilità che la pagina non contenga ALCUN record
4. $1 - (1 - \frac{1}{P})^R$ = probabilità che la pagina contenga ALMENO un record

Moltiplicando ora il punto 4 (probabilità che la pagina contenga almeno un record) per il numero delle pagine **P** otteniamo il numero medio di pagine da dover caricare

$$\Phi(R, P) = P \cdot (1 - (1 - \frac{1}{P})^R) \leq min\{R, P\}$$

Esempio: Dato un cassetto contenente calzini di P colori, ogni colore ripetuto un *infinito* numero di volte, quanti colori distinti risultano, in media, dall'estrazione di R calzioni?

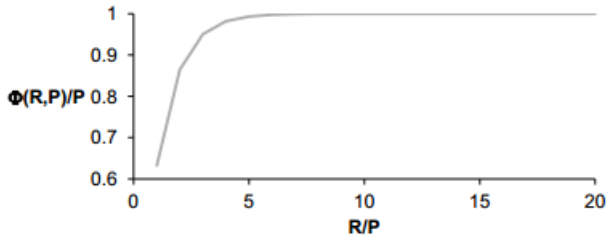
Ma nell'esempio appena descritto a cosa corrisponde il numero *infinite* di volte poi nel nostro modello di databse? Il modello appena visto (chiamato **modello di Cadenas**) assume pagine di capacità *infinita* (si noti infatti che N_t non figura da nessuna parte come argomento): un modello di questo tipo porta a *sottostimare* apprezzabilmente il valore corretto nel caso di pagine con meno di circa 10 record (cioè in realtà il valore reale medio di pagine da caricare risulterà più alto, d'altronde è ovvio che se ho pagine che contengono un numero finito di record avrò bisogno di più pagine da caricare per riuscire ad avere tutti gli R record che rispondono alla query).

Nel caso che $R = N_t$ (cioè il numero di Record sia uguale al numero di tuple), la formula restituisce un valore di P (pagine da caricare) minore, infatti:

$$\Phi(N_t, P) = P \cdot (1 - (1 - \frac{1}{P})^N \approx P \cdot (1 - e^{-\frac{N}{P}})$$

Si veda in figura (Fig. 22) l'andamento della probabiltà nel modello di Cardenas.

Figure 22:



Il modello di Cardenas, dunque, non tiene conto della capacità effettiva delle pagine ($C = \frac{N}{P}$), cosa che il **modello di Yao** invece fa. La sua derivazione considera tutti i modi possibili in cui si possono trovare allocati i **R** record richiesti sulle **P** pagine

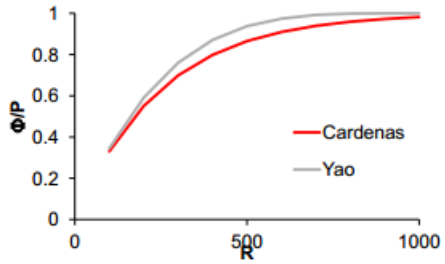
1. $\binom{N}{R}$ = numero di combinazioni possibili dei R record tra le N tuple
2. $\binom{N-C}{R}$ = numero di combinazioni possibili dei R record tra le N tuple escludendo una pagina
3. $\binom{N}{R} - \binom{N-C}{R}$ = numero di combinazioni possibili che interessano una determinata pagina
4. $1 - \dfrac{\binom{N-C}{R}}{\binom{N}{R}}$ = probabilità che una determinata pagina contenga almeno un record

Moltiplicando ora il valore 4 per il numero di pagine otteniamo il valor medio del numero di pagine accedute

$$\Phi(R, N, C) = P \cdot \left(1 - \frac{\binom{N-C}{R}}{\binom{N}{R}} \right)$$

Si veda in figura (Fig. 23) un confronto tra i modelli che mostra la probabilità di dover accedere ad una determinata pagina al crescere dei numero di Record [Dati della figura $N = 1000$, $P = 250$)

Figure 23:



Nel caso di pagine con un numero variabile di record si puà dimostrare che la formula di **Yao** sovrastima la probabilità. In generale se l'allocazione dei record non è casuale entrambi i modelli sovrastimano. Infine notiamo che se R (numero dei record) è grande, il calcolo della formula di Yao può richiedere tempi elevati

$$\Phi(R, N, C) = P \cdot (1 - \prod_{i=1}^{R} \frac{N - C - i + 1}{N - i + 1})$$

Ecco dunque che la maggior parte di volte si preferisce utilizzare il modello di Cardenas per avere un calcolo veloce. **NB** Questi modelli sono utili al *dbms* poichè deve decidere a volte se è conveniente accedere ai dati in maniera sequenziale (direttamente sulle tabelle) oppure attraverso gli indici.

Costo dell'accesso con indice Andiamo ora a vedere quale è il costo generale dell'accesso con indice. Notiamo che:

- Il Costo Totale = Costo di Indice + Costo pagine dati (cioè il costo per trovare le entry sull'indice e poi caricare le pagine dati per avere i record)
- Il Costo di Indice = costo per la prima foglia + costo per leggere tutte le foglie (cioè il costo per trovare la foglia contenente la prima entry più il costo per leggere tutte le foglie a seguire) sequenzialmente fino a trovare la prima foglia che non contiene entry utili).
  - Costo per la prima foglia = h-1
  - Numero di foglie da dover leggere = $\lceil L \cdot \frac{E_k}{K} \rceil$(cioè la frazione di foglie che soddisfano il predicato): E_k è il numero di chiavi che abbiamo nel predicato, N_Kè il numero di chiavi (distinte) totali, L è il numero totale di foglie dell'indice.
- Il Costo pagine dati = E_K volte (una per ogni chiave del predicato) la formula di Cardenas (o Yao): $E_K \cdot \Phi(\frac{N_t}{N_k}, P)$ dove $\frac{N_t}{N_k}$ rappresenta il numero di record per un certo valore di chiave.

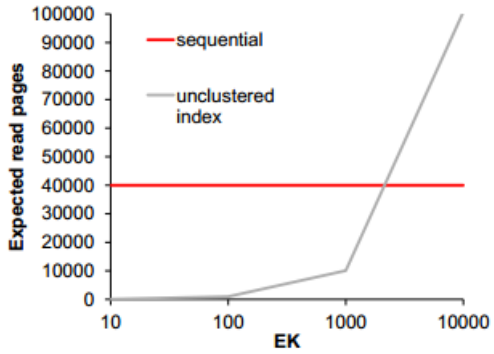
Andiamo ora a confrontare (attraverso un Esempio) il costo di accesso con indice al costo di accesso sequenziale (cioè leggere tutte le pagine che contengono la tabella) che ricordiamo essere $Costo\,Sequenziale = P$.

Esempio:

- File con $N_t = 10^6$ numero di tuple distribuiti su $P = 40000$ pagine.
- Indice unclustered (non ordinato come il file/tabella) su un campo con $N_K = 10^5$ valori di chiave (distinti), con $L = 7045$ foglie e di altezza $h = 4$ (Si noti come l'altezza dell'albero sia del tutto ininfluente).

Come si vede dalla figura (Fig.) all'aumentare del numero di chiavi che abbiamo nel predicato E_k, superando già il 10% (100.000) del numero totale di chiavi N=1.000.000, conviene decisamente accedere ai dati in maniera sequenziale.

Figure 24:



Manca ora solo controllare il costo per ricerca con intervallo $A\epsilon[x, y]$.

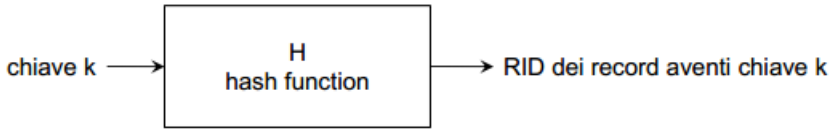
Vediamo di nuovo il costo con indice, poichè il costo di accesso sequenziale rimane sempre uguale a **P**.

- Costo totale = costo indice + costo pagine dati
- Costo indice = costo per la prima foglia + costo per leggere "sequenzialmente" le foglie (cioè il costo per trovare la foglia contenente la prima entry più il costo per leggere sequenzialmente le foglie fino a trovare la prima foglia che non contiene entry utili).
  - Costo per la prima foglia = h-1
  - Numero di foglie = $\lceil L \cdot fs \rceil$ dove fs è il fattore di selettività del predicato ($fs = \frac{y-x}{max(A)-min(A)}$)
- Costo pagine dati: $\lceil fs \cdot E_K \rceil$ volte la formula di Cardenas (o Yao):
  - $= \lceil fs \cdot E_K \rceil \cdot \Phi(\frac{N}{K}, P)$ se l'attributo non è di ordinamento
  - $= \lceil fs \cdot E_K \rceil$ se l'attributo è di ordinamento

2.5 Organizzazione hash

A differenza delle tecniche di tipo "tabellare" in cui l'associazione <chiave, RID> è mantenuta in forma esplicita, una organizzazione hash *utilizza una funzione hash* **H**, che *trasforma ogni valore di chiave in un indirizzo*. La funzione hash in breve è una funzione che data una *chiave K* restituisce automaticamente la RID dei record che hanno quel valori di chiave K (si veda figura 25). Piccola osservazione: data la figura, invece di mettere l'hash function nel box centrale avessimo messo una I di Index Table avremo comunque i RID dei record associati. Questo a sottolineare che queste sono solo strutture indirizzate a dare lo stesso risultato (RID) ma attraverso cammini d'accesso diversi.

Figure 25:



Salvo casi particolari, le funzioni *hash* non sono *iniettive*, cioè date due chiavi diverse non sempre si ottengono funzioni hash diverse ($k_1 \neq k_2 \nRightarrow H(k_1) \neq H(k_2)$). Quando questo accade si dice che si è verificata una **collisione** cioè quando date due chiavi k_1 e $k_2 \neq k_1$accade che $H(k_1) = H(k_2)$. Una funzione hash che non genera collisioni si dice *perfetta*.

Sostanzialmente che cosa è un indice hash? Detto in breve è un insieme di *pagine*. Ogni indirizzo generato dalla funzione *hash* individua una pagina logica chiamata **bucket**, quindi data una chiave k la funzione hash non fa altro che individuare l'indirizzo della pagina logica che conterrà il RID (o le RID). Se confrontato l'indice hash con un indice tabellare, il vantaggio della struttura hash è che se la funzione è efficiente, quindi calcolarla è veloce (cioè non chiede di dover leggere una pagina da disco), il costo della struttura hash è 1.

Il numero di elementi (per elementi si possono intendere sia valori di chiave se l'organizzazione è un indice ma anche i record dati se l'organizzazione è primaria) che possono essere allocati nello stesso bucket determina la *capacità* ***C*** del bucket stesso.

L'area di memoria costituita dai bucket indirizzabili dalla funzione hash è detta *area primaria*, cioè è il codominio della funzione costituito dall'insieme degli indirizzi che la funzione può dare in output. Se una chiave viene assegnata a un bucket contenente già *C chiavi* si ha un **overflow**: la presenza di overflow può richiedere, a seconda della specifica organizzazione, l'uso di un'area di memoria separata, detta appunto *area di overflow.*

Una funzione hash deve essere suriettiva (cioè deve poter raggiungere tutti gli indirizzi del codominio, altrimenti si avrebbero *bucket* inutili) e quindi generare **P** indirizzi tanti quanti sono i bucket dell'area primaria. Se il valore di **P** è, per una data organizzazione, *costante* allora l'organizzazione è detta *statica* e in

questo caso il dimensionamento dell'area primaria è parte integrante del progetto dell'organizzazione. Al contrario, se l'area primaria può espandersi e contrarsi (dunque P è dinamico, cioè il numero dei bucket può crescere e/o diminuire) per meglio adattarsi al volume effetivo dei dati da gestire allora l'organizzazione è detta *dinamica* (in caso dimanico sono necessarie più funzioni hash poichè se il codominio aumenta o diminuisce c'è bisogno di una funzione hash diversa per ogni codominio possibile). Le prime organizzazioni hash dinamiche sono nate intorno alla fine degli anni 70 mentre quelle statiche sono state sviluppate a partire dagli anni 50.

2.5.1 Caratteristiche organizzazioni hash

Sia che si tratti di una organizzazione statica, sia che sia dinamica, ci sono alcuni aspetti comuni che meritano considerazione:

- La scelta della funzione hash H
- La politica di gestione degli overflow
- La *capacità* C dei bucket dell'area primaria
- La *capacita* C_{OV} dei bucket dell'eventuale area di overflow (non necessariamente è uguale a C)
- L'utilizzazione della memoria allocata.

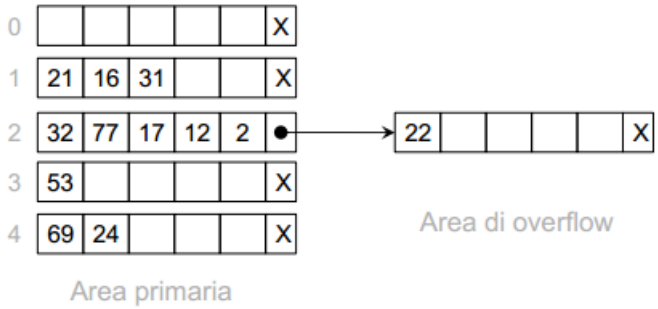
Normalmente le funzioni hash sono applicate a campi *primari* (ma l'applicazione ai campi secondari è possibile) e generalmente non preservano l'ordine (cioè non sono funzioni monotone). Non preservare l'ordine significa che se ho due valori vicini (esempio casa e case) il risultato della funzione hash potrà avere valori arbitrariamente distanti: questo incide fortemente in termini di efficienza nelle interrogazioni di intervallo.

2.5.2 Static Hashing

Mostriamo come prima cosa un semplice esempio di organizzazione hash statica (Fig. 26) in cui:

- Le chiavi sono numeri naturali
- L'area primaria consiste in P=5 (numero di bucket/pagine) di capacità C=5
- La funzione hash è siffata; $H(k_i) = k\%5$ (k *mod 5*)
- Gli overflow sono gestiti allocando, per ogni bucket dell'area primaria, uno o più bucket di overflow di capacità $\mathrm{C_{OV}}$=5 collegati a lista.

Figure 26:



Visto un semplice esempio controlliamo subito l'**analisi dei costi**.
Un file *hash* con N record, con capacità di bucket C=COV, con area primaria avente P bucket, comporta, nell'ipotesi di perfetta ripartizione dei record (cioè i record vengono ripartiti esattamente in tutti i bucket) sullo spazio dei P indirizzi, che:

- ogni indirizzo è generato $\frac{N}{P}$ volte
- ogni catena (si intende l'insieme dei bucket collegati sia in area primaria che in overflow) consiste di $\frac{N}{P \cdot C}$ bucket

Il costo di ricerca di un record risulta pertanto propozionale a $\frac{N}{P \cdot C}$ poichè, data una chiave, il RID del record che cerchiamo potrebbe trovarsi all'ultimo bucket situato nell'area di overflow. Esempio: $N = 10^6$, $C = 10$, $P = 25000$ Una ricerca (con successo) accede in media a 2 bucket (ho eseguito il calcolo di $\frac{N}{P \cdot C}$) e nel caso peggiore a 4 bucket (l'ultimo in area di overflow). Questo risulta essere anche il numero di operazioni I/O se si assume che ogni bucket può essere letto con una sola operazione.

2.5.3 Funzioni hash

Una funzione hash è *una trasformazione* (suriettiva) *dallo spazio K delle chiavi allo spazio* $\{0, \ldots, P-1\}$ *degli indirizzi*. L'ipotesi che un arbitrario sottoinsieme di K si ripartisca su P indirizzi in maniera perfettamente omogenea è una pura astrazione, di scarsa utilità per analizzare le prestazioni ottenibili dalle diverse organizzazioni hash.

Il caso ideale rispetto al quale è ragionevole confrontare una speifica funzione hash H è quello di distribuzione uniforme sullo spazio degli indirizzi, in cui, per ogni sotto insieme di K chiavi, ognuno dei P indirizzi ha la stessa probabilità $\frac{1}{P}$ di essere generato. Controlliamo dunque la funzione hash con distribuzione uniforme.

Funzione hash con distribuzione uniforme Nel caso ideale il numero di chiavi X_j assegnate al *j-esimo* bucket seguen una distribuzione binomiale

$$P(X_j = x_j) = \binom{N}{x_j} \cdot (\frac{1}{P})^{x_j} \cdot (1 - \frac{1}{P})^{N - x_j}$$

con valor medio e varianza ^{2} dati da $= \frac{N}{P}$ e $^2 = \frac{N}{P} \cdot (1 - \frac{1}{P})$ in cui né nè σ^2 dipendono dallo specifico bucket. Come si nota per $P \gg 1$ il rapporto $\frac{\sigma}{\sqrt{\mu}} \sim 1$.

Qualità di una funzione hash Nel caso di funzioni hash *reali* le prestazioni variano al variare dello specifico set di chiavi. Ad esempio la funzione hash $H(k_i) = k_i \% P$ è una "buona" funzione ma nel caso il set di chiavi sia $\{0, P, 2P, 3P, \ldots, N \cdot P\}$ alloca tutte le chiavi nel bucket 0. **NB** Per ogni funzione hash esiste un set di chiavi che dà luogo a pessime prestazioni nel caso peggiore. Nel caso "medio" tuttavia, considerando arbitrari sottoinsiemi dell'insieme K di chiavi e file dati reali, si osserva che le diverse funzioni hash si comportano effettivamente in modo diverso.

Un criterio adeguato di valutazione di una funzione H, riferito ad un particolare insieme di chiavi K, è dato dall'analisi della sua **degenerazione** $\frac{\sigma}{\sqrt{\mu}}$ dove

$$\mu = \sum_{j=0}^{P-1} \frac{x_j}{P} = \frac{N}{P}$$

$$\sigma^2 = \sum_{j=0}^{P-1} \frac{(x_j - \mu)^2}{P}$$

sono calcolati su tutti i **P** bucket e x_j è il numero di record osservato nel j-esimo bucket. Quanto più bassa è la degenerazione tanto migliore è il comportamento della funzione hash.

Esempi Funzioni Hash Elencheremo di seguito alcuni fra i più comuni esempi di funzione hash.

Mid Square La chiave k viene moltiplicata per sé stessa, viene dunque estratto un numero di cifre centrali pari a quelle di P-1, il numero ottenuto viene dunque normalizzato a P.

Ad esempio con la chiave $k = 145142$ si ottiene $k^2 = 145142^2 = 21066200164$ e con numero di bucket $P = 8000$ la normalizzazione produce $\lfloor 6620 \cdot 0.8 \rfloor = 5296$.

Shifting La chiave k viene suddivisa in un certo numero di sotto parti, ognuna costituita da un numero di cifre pari a P-1, infine si sommano le parti e si normalizza il risultato.

Ad esempio con la chiave $k = 14514387$ si ottengono le sotto parti $k_1 = 387$, $k_2 = 514$ e $k_3 = 14$, se ne fa la somma $k_1 + k_2 + k_3 = 387 + 514 + 14 = 915$ e con un numero di bucket $P = 800$ la normalizzazione produce $\lfloor 915 \cdot 0.8 \rfloor = 732$

Folding La chiave k viene suddivisa come allo stesso modo in cui accadeva nello **shifting**, però ora le parti vengono "ripiegate" (si immagini di avere un foglio e piegare verso l'interno man mano il foglio stesso) e sommate, infine si normalizza il risultato.

Ad esempio con $k = 14514387$ si ottengono le sotto parti $k_1 = 783$ (prima "ripiegatura", come si vede è la cifra 387 scritta al contrario), $k_2 = 514$ (la seconda non viene ripiegata, proprio come in un foglio accade che la seconda parte permane integra) e $k_3 = 41$ (ultima "ripiegatura", è l'opposto di 14); se ne fa la somma $k_1 + k_2 + k_3 = 783 + 514 + 41 = 1338$ e con un numero di bucket $P = 800$ la normalizzazione produce $1338\%800 = 538 \longrightarrow \lfloor 538 \cdot 0.8 \rfloor = 430$.

Divisione La chiave *numerica* k viene divisa per un numero Q e l'indirizzo è ottenuto considerando il resto (il *mod* o %), cioè $H(k) = k\%Q = k \, mod \, Q$. Per la scelta di Q ci sono alcune linee guida pratiche:

- Q è il più grande *numero primo* minore o uguale a P (numero di bucket)
- Q è non primo, minore o uguale a P, con nessun *fattore primo* minore di 20
- Se $Q < P$ si deve porre $P = Q$ per non perdere la suriettività della funzione hash (poichè tutti i bucket superiori a $P - Q$ risulterebbero non raggiungibili).

2.5.4 Chiavi alfanumerica

Fino ad ora abbiamo sempre pensato e usato chiavi *numeriche*: questa semplificazione però non è attuabile in pratica, infatti, come sappiamo, gli attributi alfanumerici fanno da padroni nei database. Si rivela dunque necessario trattare le stringhe *alfanumeriche*.

Una chiave *alfanumerica* ha bisogno di una fase preliminare di conversione per essere poi realmente gestita attraverso funzioni hash.
Uno dei metodi più comuni è quello di stabilire

- Un alfabeto A a cui appertengono i caratterei delle stringhe
- Una funzione **biettiva** *ord*( ) che associa ad ogni elemento dell'alfabeto un intero appartenente all'intervallo $[1, |A|]$
- Una *base b* di conversione

Una stringa $S = s_{n-1}, \ldots, s_i, \ldots, s_0$ viene dunque convertita in una *chiave numerica* nel seguente modo

$$k(S) = \sum_{i=0}^{n-1} ord(s_i) \cdot b^i$$

Facciamo un semplice esempio.
Scegliamo come alfabeto $A = \{a, b, \ldots, z\}$ di cardinalità $|A| = 26$ a cui associamo la nostra funzione *biettiva* $ord(\ )$ a valori nell'intervallo [1, 26]. Scegliamo come base $b = 32$ e vediamo come ottenere la nostra chiave numerica della stringa "indice":

$$k(\text{"}indice\text{"}) = 5 \cdot 32^5 + 14 \cdot 32^4 + 4 \cdot 32^3 + 9 \cdot 32^2 + 3 \cdot 32^1 + 5 \cdot 32^0 = 316810341$$

Esistono anche metodi più semplici per la conversione di stringhe alfanumeriche che non fanno uso di alcuna *base* quali ad esempio

$$k(S) = \sum_{i=0}^{n-1} ord(s_i)$$

in cui $ord(\ )$ ricopre lo stesso ruolo prima descritto. Questi metodi sono più facili da utilizzare ma funzionano meno bene, in quanto generano la stessa chiave numerica a partire da anagrammi diversi di una stessa stringa (ad esempio *caso* e *cosa* produrranno la stesssa chiave numerica).

Scelta della base Se si dovesse scegliere come funzione *hash* il metodo della **divisione** si riscontrano seri problemi se la *base* b ha *fattori primi in comune* con **P** (numero dei bucket). Per chiarire facciamo subito un esempio. $A = \{a, b, \ldots, z\}$, $b = 32$, $P = 512$. Come si vede 32 e 512 hanno uno stesso fattore primo in comune (il 2). Calcoliamo ora le chiavi numeriche di "*folder*" e di "*primer*".

I valori di ord( ) delle due parole producono il vettore di coefficenti

$$ord(\text{"}folder\text{"}) = (6, 15, 12, 4, 5, 18)$$

$$ord(\text{"}primer\text{"}) = (16, 18, 9, 13, 5, 18)$$

e calcoliamo ora anche i valori di chiave numeriche

$$k(\text{"}folder\text{"}) = 215'452'722$$

$$k(\text{"}primer\text{"}) = 556'053'682$$

e infine calcoliamo la funzione hash dei valori numerici

$$H(215'452'722) = 178$$

$$H(556'053'682) = 178$$

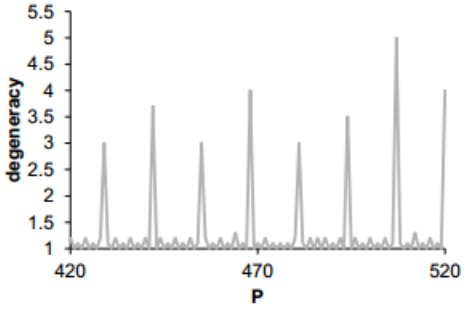
In questo cosa, attraverso alcune proprietà dell'operatore %, si può notare che il valore di $H(k(S))$ è determinato solo dagli ultimi due caratteri. Per capire perchè insorge questo tipo di problema è necessario rifarsi ad una serie di proprietà dell'operatore modulo (% o mod). Il caso più semplice da considerare è quello in cui **P** (numero di bucket) è un multiplo di b (base), ovvero $P = \alpha \cdot b$ con α valore intero. Esisterà un valore y tale per cui $b^y \,\%\, (\alpha \cdot b) = 0$. Poichè per l'operatore % vale la seguente proprietà (utile per il calcolo di k(S))

$$H(k(S)) = \left[\sum_{i=0}^{n-1} ord(s_i) \cdot b^i\right] \%(\alpha \cdot b) = \left(\sum_{i=0}^{n-1} \left[\left(ord(s_i) \cdot b^i\right) \%(\alpha \cdot b)\right]\right) \%(\alpha \cdot b)$$

i caratteri da s_{n-1} a s_y danno contributo uguale a zero al valore $H(k(S))$ (nell'esempio prima mostrato $y = 2$) e quindi la stringa "utile" è lunga solo y caratteri. Con base $b = 26$ si ganno problemi quando P ha come fattori 13 e 2.

Viene mostrato di seguito un esperimento eseguito da *Mullin* nel 1991 facendo uso come set di chiavi k tutte le parole di 6 caratteri costruite sull'alfabeto $A = \{a, b, \ldots, z\}$ usate dallo spelling checker di Unix. La figura (Fig. 27) si riferisce al caso di *base* b=26 con P che varia tra 420 e 520.

Figure 27:



I picchi della distribuzione si hanno per valori di P multipli di 13. In particolare il massimo si ottiene per $P = 507 = 13 \cdot 13 \cdot 3$.

2.5.5 Parametri

Fattore di caricamento Supponiamo di aver scelto la nostra funzione hash statica e di cominciare a pensare quanto far grande la nostra organizzazione ad hash: una prima idea potrebbe essere quella, una volta avuta la stima del numero N di record, di avere il numero di bucket P in modo tale che i record N vadano ad occupare "completamente" i nostri bucket. In realtà potremmo pensare di lasciare "un po' di spazio" disponibile nei nostri bucket, cioè di realizzare un'area primaria più grande del necessario, per tentare di evitare l'overflow infatti potrebbe convenire avere dello spazio vuoto in più piuttosto che rischiare l'immediato overflow.

Data una stima del numero N di record da gestire e fissata la capacità C dei bucket, la scelta di un determinato *fattore di caricamento d* determina il numero di bucket **P** in area primaria, infatti al diminuire di d diminuisce la percentuale di record in overflow: di conseguenza non è consigliabile utilizzare fattori di caricamento elevati. Valori tipici che rappresentano un buon compromesso tra utilizzazione della memoria e costi di esercuzioni delle operazioni si hanno nell'intervallo [0.7, 0.8].

Capacità dei bucket È evidente che avere una capacità C dei bucket tale da richiedere più operazioni di I/O per il trasferimento di un singolo bucket non comporta alcun vantaggio. Il motivo per cui invece è conveniente avere bucket di capacità $C > 1$ è essenzialmente legato alla relazione esistente tra C e tra la percentuale di record costretti ad andare in *overflow*. Sotto le ipotesi di una funzione hash ideale e di gestione degli overflow in un'area separata, all'aumentare di C (e a parità di fattore di caricamento d) la percentuale di record in overflow diminuisce (sperimentalmente si vede che il risultato è valido anche nel caso di funzioni hash non ideali), quindi si potrebbe pensare anche di avere C maggiore anche della grandezza della pagina con la supposizione però di poter scrivere i blocchi in maniera contigua (in modo da evitare il tempo di latenza).

Poichè un aumento del numero di record in *overflow* tende a deteriorare le prestazioni è consigliabe lavorare con bucket di capacità C *massima* soggetti però ai seguenti vincoli:

- C deve essere tale che la lettura di un bucket deve comportare ad una singola richiesta di I/O (quindi per ovviare al problema dei blocchi contigui)
- C deve essere tale che il trasferimento di un bucket di capacità C deve poter avvenire in un tempo minore rispetto al trasferimento di due bucket di capacità $\bar{C} < C$

2.5.6 Overflow

Ci occupiamo ora della gestione degli *overflow* causati dalla presenza di un numero di elementi nel bucket j-esimo di *capacità C* superiore al *fattore di caricamento d.*

Numero medio di overflow Il numero di volte che una funzione di *hash* genera lo stesso indirizzo j è riconducibile a una variabile aleatoria X_j con *distribuzione binomiale.* Il numero medio di overflow del j-esimo bucket è

$$OV_j(C) = \sum_{x_j=C+1}^{N} (x_j - C) \cdot Pr\{X_j = x_j\}$$

che non dipende dal j-esimo bucket specifico ma dalla probabilità che la funzione hash scelta generi l'indirizzo esattamente x_j volte. Per brevità da ora in poi scriveremo Pr(x) al posto di $Pr\{X_j = x_j\}$.

Distribuzione degli overflow Il numero totale di *overflow* si ottiene sommando il numero medio degli overflow prima citato su tutti i P bucket, dunque

$$OV(C) = \sum_{j=0}^{P-1} OV_j(C) = P \cdot \sum_{x_j=C+1}^{N} [(x_j - C) \cdot Pr(x)]$$

Per valori elevati di N (numero dei record) e P (numero dei bucket) è possibile approssimare la distribuzione binomiale con la distribuzione di *Poisson*:

$$Pr(x) = \binom{N}{x} \cdot \left(\frac{1}{P}\right)^x \cdot \left(1 - \frac{1}{P}\right)^{N-x} \approx \left(\frac{N}{P}\right)^x \frac{e^{-\frac{N}{P}}}{x!}$$

Essendo inoltre $P = \frac{N}{C \cdot d}$ otteniamo sostituendo

$$Pr(x) \approx (C \cdot d)^x \cdot \frac{e^{-C \cdot d}}{x!}$$

$C \cdot d$ denota il numero medio di elementi in un bucket.

Numero totale di overflow Il numero totale di *overflow* si ottiene quindi sostituendo alle formule le approssimazioni prima scritte ottenendo

$$OV(C) \approx P \cdot \sum_{x=C+1}^{N} \left[(x - C) \cdot \frac{(C \cdot d)^x \cdot e^{-(C \cdot d)}}{x!}\right]$$

cambiando variabile in $i = x - C$ otteniamo

$$OV(C) \approx P \cdot \frac{(C \cdot d)^{C+1} \cdot e^{-(C \cdot d)}}{C!} \cdot \sum_{i=1}^{N-C} \frac{i \cdot C^{i-1} \cdot d^{i-1}}{(C+1) \cdot (C+2) \cdot \ldots \cdot (C+i)}$$

e ricordando nuovamente che $P = \frac{N}{C \cdot d}$ e sostituendo otteniamo

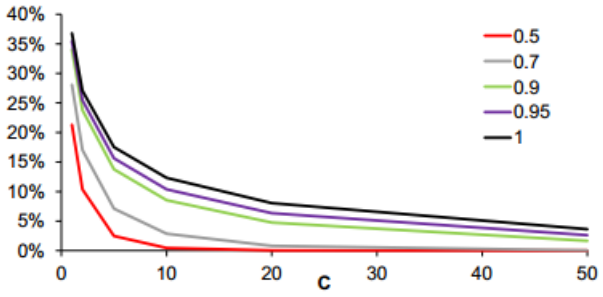
$$OV(C) \approx N \cdot \frac{(C \cdot d)^C \cdot e^{-(C \cdot d)}}{C!} \cdot f(C, d)$$

dove $f(C, d)$ è la sommatoria presente nella formula precedente (che si nota essere una funzione solo della capacità C e del fattore di riempimento d)

Mostriamo dunque un grafico (Fig. 28) in cui viene raffigurato l'andamento degli overflow al variare della *capacità C* e del *fattore di riempimento d.*

Figure 28:

C\d	0.5	0.7	0.9	0.95	1
1	0.213061	0.280836	0.340633	0.354464	0.367879
2	0.103638	0.170307	0.237853	0.254377	0.270669
5	0.02478	0.071143	0.137768	0.156336	0.17531
10	0.004437	0.028736	0.085344	0.103778	0.123244
20	0.000278	0.008014	0.047641	0.063497	0.080492
50	2.51E-07	0.000496	0.016561	0.026191	0.03622



2.5.7 Gestione degli overflow

I metodi studiati per la gestione degli *overflow* mirano a ridurre al minimo gli accessi a bucket necessari per reperire il record cercato. Esistono sostanzialmente due strategie:

1. Metodi di **concatenamento** (chaining) i quali:
   (a) Utilizzano puntatori (per sapere dove reperire i bucket overflow)
   (b) Possono usare o meno l'area di overflow
2. Metodi di **indirizzamento aperto** (open addressing) i quali:
   (a) Non fanno uso di puntatori, ma utilizzano una famiglia di funzioni hash
   (b) Usano bucket in area primaria per memorizzare i record in overflow

Metodi di concatenamento I metodi di concatenamento possono differenziarsi a seconda dell'utilizzo o meno dell'area di overflow. Guardiamo nel dettaglio entrambi i metodi.

Concatenamento in area primaria Il concatenamento in area primaria può avvenire utilizzando due tecniche

1. **Liste separate**

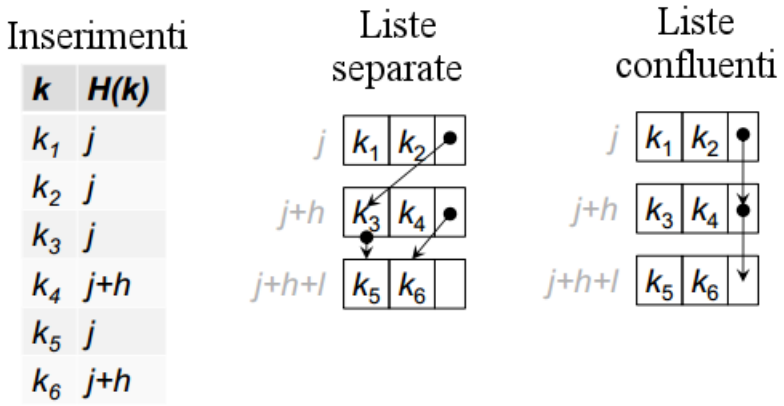
   Se il bucket j va in *overflow*, si inserisce il nuovo record nel primo bucket **non pieno** successivo a j. In questo caso è necessario che sia i record in *overflow* sia i record non in overflow siano collegati a lista (in modo separato), così da riuscire a distinguere, in un bucket, se il record presente nel bucket stesso fa parte del bucket poichè indirizzato dalla funzione hash oppure se la sua presenza è dovuta a un bucket precedente andato in oveflow

2. **Liste confluenti** (coalesced chaining)

   A differenza delle *liste separate* nelle *liste confluenti* si utilizza un solo puntatore per ogni bucket (e non più per ogni record). Se ora un bucket j va in overflow, esso sceglierà ancora il primo bucket libero successivo ad esso per mandare i record, ma avrà un puntatore a questo nuovo bucket $j+h$, il quale, se pure lui si trova in overflow, punterà ad un nuovo bucket $j+h+l$ e così via, creando dunque una *lista confluente*.

Di seguito (Fig. 29) un esempio di liste separate e liste confluenti.

Figure 29:



Il metodo a liste confluenti semplifica la gestione dei puntatori ma peggiora la prestazioni. Si prenda l'esempio e mettiamo di voler cercare una chiave, non presente nell'indice, che porti al bucket *j:* nelle liste separate devo proseguire fintanto che i puntatori dei record non finiscono, mentre nella lista confluente dovrò seguire la lista fintanto che ho bucket collegati (e fino a qui le prestazioni sono simili), ma cosa accade se cerco una chiave che porta al bucket $j+h$? Nel

caso di liste separate ancora una volta proseguo con i puntatori dei record fintanto che non trovo il record che sto cercando ed essendo i record del bucket collegati posso fermarmi al bucket stesso in caso di record trovato, mentre nelle liste confluenti, essendo che ora sono i bucket ad essere connessi da lista, dovrò andar a scandire ugualmente i bucket successivi, azione che potrebbe non essere necessaria ma obbligata.

Concatenamento in area di overflow Come detto in precedenza, la capacità dei blocchi nell'area di overflow C_{OV} può essere diversa dalla capacità dei bucket dell'area primaria C. In genere $C_{OV} < C$ per evitare sprechi: infatti solitamente si tenta di ridurre il meno possibile l'overflow, di conseguenza la capacità dei blocchi in overflow può essere minore e per lo stesso motivo si può decidere di utilizzare *liste confluenti*. Ovviamente, in caso di overflow di un bucket di overflow, si otterà una lista confluente di bucket di overflow.

2.5.8 Gestione Overflow - Open Addressing

Dedichiamo ora un sottoparagrafo alla tecnica del open addressing, poichè come si vedrà, richiede alcuni particolari accorgimenti.

A differenza del metodo della gestione a liste, dove a partire dall'indirizzo originale seguo una lista attraverso puntatori, nei metodi a *indirizzamento aperto* (open addressing) a ogni valore di chiave k_i viene associata **una sequenza di indirizzi** $H_0(k_i)$, $H_1(k_i)$, ..., $H_l(k_i)$ (ho creato cioè una "lista" attraverso una famiglia di funzioni *hash*). Quando si inserisce una nuova chiave k_i si provano tutti gli indirizzi generati da ogni funzione hash $H_0(k_i)$, $H_1(k_i)$, ..., $H_l(k_i)$ fino a quando non si trova l'indirizzo di un bucket *non pieno*.

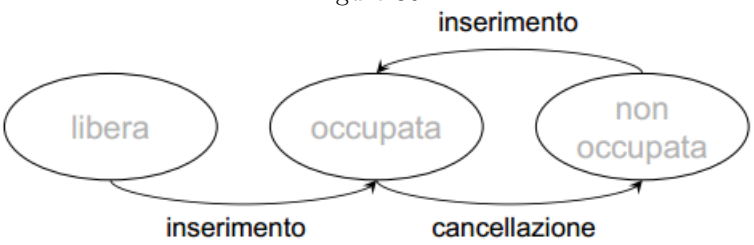
Ricerca Per cercare k_i occorre cercare in tutti gli indirizzi associati $H_0(k_i)$, ..., $H_l(k_i)$ fino a quando:

- si trova k_i (ricerca con successo)
- si trova un bucket *non pieno* (ricerca con insuccesso).

La ricerca infatti può concludersi quando viene trovato un bucket non pieno poichè l'inserimento cerca il primo bucket non pieno in cui inserire la chiave k_i e di conseguenza se raggiungiamo un bucket non pieno e k_i non è stata ancora trovata vuol dire che la chiave non è presente.

Nei metodi a indirizzamento aperto occorre quindi prestare particolare *attenzione* a come si operano le *cancellazioni*: infatti se si cancella un record presente in un bucket che risultava *pieno* esso diventerà *non pieno* "interrompendo" dunque la ricerca.

Figure 30:



Cancellazione Per la cancellazione è utile introdurre un nuovo "stato" alle posizioni all'interno del bucket, infatti la semplice distinzione pieno e non pieno del bucket non è più esaustiva per la ricerca. Ecco allora che le "posizioni" di un record all'interno di un bucket possono essere marcate come *libere*, *occupate*, *non occupate.* La posizione di un record cancellato viene quindi marcata come *non occupata* e può essere riutilizzata (e essere marcata nuovamente come *occupata*) in seguito a nuovi inserimenti. Un bucket risulta *pieno* **se e solo se** tutte le posizioni all'interno del bucket sono **occupate.** Si noti che all'atto dell'inserimento sarà dunque preferibile andare a occupare posizioni "non occupate" piuttosto che posizioni "libere".

Famiglia di Hash Come si creano le famiglie di funzioni hash necessarie ad indirizzare tutti i bucket presenti in memoria principale?

La prima idea è la **scansione lineare** (linear probing). Ad ogni passo l'indirizzo risultante della j-esima funzione hash H è dalla somma del risultato della funzione hash *j-1*-esima più un **passo costante** ***s*** $H_j(k_i) = [H_{j-1}(k_i) + s]\%P$ che con una semplice trasformazione risulta essere $H_j(k_i) = [H(k_i) + s \cdot j]\%P$. Si noti che per generare tutti i P indirizzi occorre che il passo s sia *cooprimo* con P (cioè che non abbiano divisori comuni) altrimenti verrebbero generati solo $\frac{P}{MCD(P,s)}$ indirizzi. Ad esempio: P=10, s=4, MCD(10,4)=2, gli indirizzi generabili per le chiavi sono 5, infatti $k_i = 3$ genera gli indirizzi 3, 7, 1, 5, 9, 3.

In un **clustering primario** risulta essere presente un altro problema, il cosiddetto *addensamento di record*: se un bucket j va in overflow è molto probabile che il cluster (bucket) j+s vada in overflow, quindi che il cluster j+2s vada in overflow e così via. Ad esempio con P=31 e s=3 vengono generati i seguenti indirizzi:

- $k_i = 1234 \rightarrow H_0(1234) = 1234\%31 = 25 \rightarrow H_1(25 + 3) = 28 \rightarrow 0 \rightarrow 3 \rightarrow 6 \rightarrow 9 \rightarrow \ldots$
- $k_i = 245 \rightarrow H_0(245) = 245\%31 = 28 \rightarrow H_1(28 + 3) = 0 \rightarrow 3 \rightarrow 6 \rightarrow 9 \rightarrow \ldots$

Il problema è dovuto alla linearità del *passo di scansione* ***s***. Per risolvere il problema si può pensare ad introdurre la **scansione quadratica.** Nella scan-

sione quadratica il passo non è una costante s, ma una funzione lineare in j. Ad ogni passo l'indirizzo è aumentato di un *passo lineare* $a + b \cdot (2j - 1)$ e dunque la funzione hash j-esima è data da $H_j(k_i) = [H_{j-1} + a + b \cdot (2j - 1)]\%P$ che con una semplice trasformazione risulta essere $H_j(k_i) = [H_j(k_i) + a \cdot j + b \cdot j^2]\%P$. Vediamo con un esempio come la scansione quadratica sia riuscita ad evitare l'addensamento di record. Esempio: P=31, a=3, b=5 indirizzi generati:

- $k_i = 1234 \rightarrow H_0(1234) = 1234\%31 = 25 \rightarrow H_1(25 + 3 \cdot 1 + 5 \cdot 1) = 2 \rightarrow 20 \rightarrow 17 \rightarrow 24 \rightarrow \ldots$
- $k_i = 245 \rightarrow H_0(245) = 245\%31 = 28 \rightarrow H_1(28 + 3 \cdot 1 + 5 \cdot 1) = 5 \rightarrow 23 \rightarrow 20 \rightarrow 27 \rightarrow \ldots$

e in effeti le liste non confluiscono (si veda il 20 che produce indirizzi diversi).

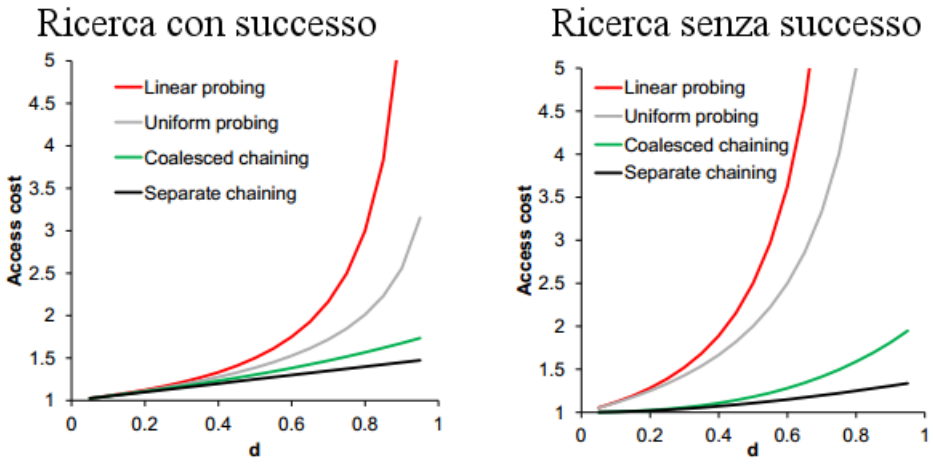
Rimane però ancora il problema del **clustering secondario** dovuto a chiavi aventi lo stesso "primo" bucket. Anche questo caso è un problema risolvibile, grazie alla tecnica del **double hashing** la quale consisnte di generare la famiglia di funzioni hash $H_0(k_i)$, ..., $H_l(k_i)$ a partire da *due funzioni hash H' e H"*. In questo caso le sequenze di indirizzi sonon date da:

- $H_0(k_i) = H'(k_i)$
- $H_j(k_i) = [H_{j-1}(k_i) + H"(k_i)]\%P$ (se $j > 0$)

Adesso due chiavi generano la stessa sequenza di indirizzi **se e solo se** entrambe collidono sia con *H'* sia con *H"* (molto improbabile). Si noti come la *double hashing* produce una *scansione lineare* (non è quadratica) quindi a passo costante, però a differenza della semplice scansione lineare il passo s varia da chiave a chiave. La tecnica del double hashing ha come effetto collaterale quello di produrre una forte variabilità degli indirizzi generati, in maniera dipendente dal valore di $H"(k_i)$. Considerando l'effettiva allocazione dei bucket in memoria secondaria ciò può appesantire *notevolmente* le operazioni di I/O poichè bucket successivi (il primo ed il secondo ad esempio) si ritrovano "lontani" andando ad aumentare parecchio il tempo di latenza (tempo impiegato dalla testina per muoversi). Nonostante questo svantaggio, il *double hashing* riesce però ad approssimare abbastanza bene il nostro caso ideale di "**hash uniforme**" in cui ogni indirizzo ha la stessa probabilità di essere generato al j-esimo passo.

Viene infine mostrato un confronto (Fig. 31) tra le varie tecniche finora mostrate.

Figure 31:



2.5.9 Hashing Dinamico

Finora abbiamo discusso di organizzazioni hash statiche, in cui l'allocazione della memoria è effettuata al momento della progettazione iniziale. Al momento di progettazione vengono scelti dunque il numero dei bucket (P), la loro capacità (C) e il fattore di caricamento (d) basandosi sulla stima del numero di record (N) che si dovranno gestire. Si noti che essendoci basati su una stima, se essa è in eccesso allora lo spazio risulterà poco utilizzato, mentre se è in difetto si dovrà lavorare con un alto fattore di caricamento (e conseguentemente elevati costi di accessi): se inoltre gli *overflow* vengono gestiti in area primaria bisogna ricordarsi del vincolo $d \leq 1$. Cosa succede inoltre se il mio dataset (insieme delle chiavi) aumenta? I parametri prima imposti che si basavano sulla stima di N ora risulterebbero errati proprio perchè N è aumentato. Introduciamo dunque l'organizzazione ad *hashing dinamico*. **NB** L'hashing statico raggiunge la sua ottimalità se utilizzato su una "tabella" che sappiamo non andrà a cambiare, questo vuol dire che utilizzeremo l'hashing statico come strumento di appoggio alla risoluzione di query, poichè alle interrogazioni sul server le tabelle sono già "date" e non possono variare (ad esempio sarà molto utile nella risoluzione di *join*).

Le tecniche di *hashing dinamico* adattano l'allocazione dell'area primaria in base al numero di tuple attuali. Esse si possono raggruppare in due gruppi:

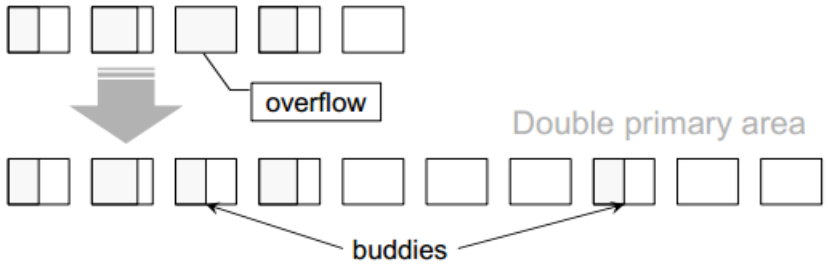
- **Con directory**
  - Virtual hashing
  - Dynamic hashing
  - Extendible hashing
- **Senza directory**

– Linear hashing
– Spiral hashing

Verranno trattati in maniera specifica l'**extendible hashing**, il **virtual hashing** e il **dynamic hashing** e il **linear hashing** dando alcuni concetti generali delle restanti tecniche.

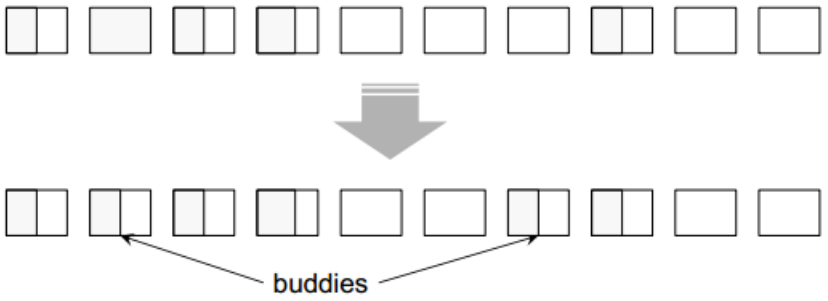
Virtual hashing L'idea su cui si basa il *virtual hashing* (Litwin '78) è di *raddoppiare l'area primaria* quando si verifica un *overflow* di un bucket (e ad esso non è associato alcun *bucket buddy* 'compagno' sul quale ridistribuire i record). La directory terrà memoria di quali *bucket* sono effettivamenti usati e quali no. Quando si verifica un *overflow* di un bucket si ridistribuiscono i record tra il bcket saturo e il suo *buddy* facendo uso di una nuova funzione hash: in pratica si esegue lo *split* del bucket saturo (Fig. 32).

Figure 32:



Se successivamente un altro bucket nell'area primaria originale va in overflow e il suo *buddy* non è ancora in uso, il bucket buddy inizia ad essere utilizzato (e quindi viene segnato come in uso) e vengono ridistribuiti i record tra il bucket e il bucket *buddy* (Fig. 33).

Figure 33:



Poichè, in un certo istante, solo alcuni *buddy* sono effettivamente in uso è necessario far uso di una struttura ausiliaria, *la directory*, che permetta di determinare se occorre utilizzare la vecchia o la nuova funzione *hash*. Sostantialmente la directory è un semplice vettore V di bit (0 o 1) in cui $V[i] = 1$ *se e solo se* il bucket i-esimo è in uso.

Come viene inizializzata l'area primaria? Vengono allocati P_0 bucket di capacità C. Si usa una funzione hash H_0 che restituisca valori nell'intervallo $[0, P_0 - 1]$. Si utilizza un contatore I che determina il numero dei raddoppi eseguiti, quindi si inizializza intanto a 0 $I = 0$. Si crea dunque un vettorei binario V di dimensione pari a P_0 ponendo tutti gli elementi al valore 1 (poichè assumiamo che tutti i bucket dell'originale area primaria vengano usati). Dopo I raddoppi l'area primaria conterrà un numero di bucket P pari a $P = 2^I \cdot P_0$ dove ogni j-esimo bucket (con $0 \leq j \leq 2^{I-1} \cdot P_0 - 1$ cioè la prima metà dell'area di memoria) ha associato il suo bucket *buddy* all'indirizzo $j + 2^{I-1} \cdot P_0$.

Controlliamo ora come avviene uno *split* di un bucket j.
Se $I = 0$ (cioè non è stato fatto ancora alcun raddoppio) oppure $I > 0$ ma il bucket buddy del nostro bucket j-esimo risulta essere già in uso (dunque $I > 0$ ma $V[j + 2^{I-1} \cdot P_0] = 1$) o non esiste (dunque $I > 0$ ma $j \geq 2^{I-1} \cdot P_0 - 1$) allora:

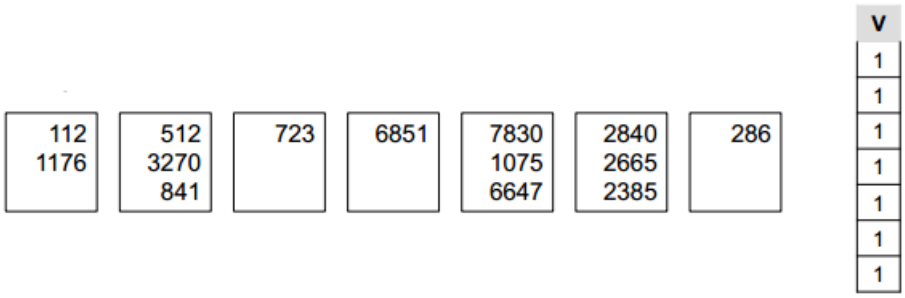
1. Viene incrementato il valore di I$\rightarrow I++$
2. Si raddoppia l'area primaria e il vettore V
3. I nuovi elementi di V vengono impostati a 0 eccetto per $V[j + 2^{I-1} \cdot P_0]$ (cioè il bucket buddy del bucket j-esimo)
4. Si crea una nuova funzione H_I a valori in $[0, 2^I \cdot P_0 - 1]$
5. Si ridistribuiscono le chiavi del bucket j utilizzando la nuova funzione H_I

Se invece il *bucket buddy* del nostro bucket j-esimo esiste ma non è in uso (cioè $I > 0$ e $V[j + 2^{I-1} \cdot P_0] = 0$) allora

1. Viene impostato a 1 il valore di utilizzo del bucket buddy $\rightarrow V[j + 2^{I-1} \cdot P_0] = 1$
2. Si ridistribuiscono le chiavi del bucket j utilizzando la funzione H_I (che sicuramente esiste già)

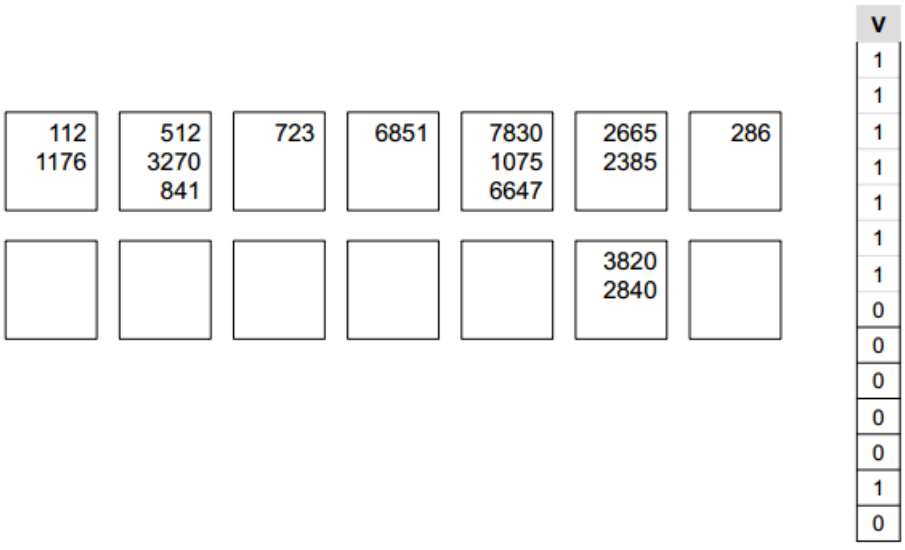
Facciamo un esempio. Immaginiamo di avere numero di bucket iniziali $P_0 = 7$ con capacità $C = 3$ riempiti come mostrato in figura (Fig. 34)

Figure 34:



La famiglia di funzioni hash H_I sia siffata $H_I(k) = k\%(2^I \cdot P_0)$. Supponiamo di voler inserire la chiave $k = 3820$ la cui funzione hash $H_0(3820) = 5$ (si noti che non è stato fatto ancora alcun raddoppio quindi $I = 0$). Il bucket 5 va in overflow (siamo sempre 0-based). Bisogna dunque effettuare un raddoppio dell'area di memoria e del vettore V impostando a 0 tutti i nuovi valori eccetto quello del bucket buddy, incrementare di 1 il valore di I ($P_1 = 14$ e $I = 1$) e ridistribuire le chiavi del bucket 5, tra il bucket 5 stesso e il suo buddy 12 facendo uso della nuova funzione hash $H_1(k) = k\%14$. In figura viene mostrata la nuova situazione (Fig. 35).

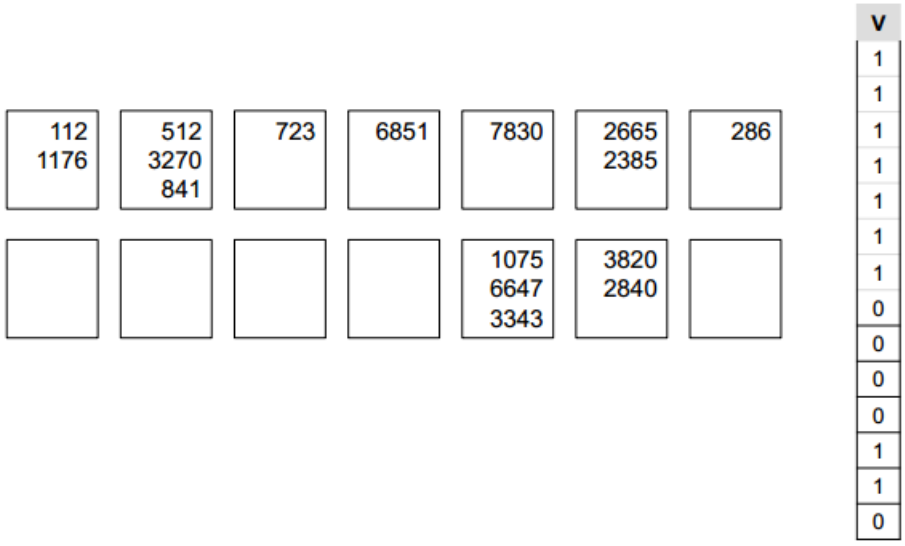
Figure 35:



Se ora volessi inserire la chiave 3343, si ha $H_1(3343) = 11$ ma il bucket 11 ha $V[11] = 0$ (dunque non in uso), conseguentemente bisogna applicare la funzione hash precedente $H_0(3343) = 4$. Il bucket 4 risulta però saturo quindi bisogna eseguire lo split: in questo caso però l'area può non raddoppiare perchè il suo

bucket buddy esiste ed esso risulta non essere ancora utilizzato. Di conseguenza viene impostato a 1 il valore di $V[11] = 1$ e vengono ridistribuite le chiavi del bucket 4 utilizzando la nuova funzione H_1. In figura (Fig. 36) viene mostrata la nuova situazione.

Figure 36:



Manca ancora da capire come funziona la ricerca di una chiave. Per cercare un valore di chiave è necessario sapere con quale funzione hash è stato allocato il valore stesso. Il vettore V è sufficiente allo scopo. Il seguente algoritmo ricorsivo fornisce l'indirizzo del bucket in cui potrebbe trovarsi (magari non esiste) la chiave cercata.

`Address(k, I)`

Input: Chiave k da cercare, livello I in cui cercare

Output: Indirizzo del bucket al livello I in cui trovare k

```
if (I<0) then la chiave non esiste
else if (V[H_I(k)]=1 return H_I(k)
else return Address(k, I-1)
```

Il virtual hashing richiede una *serie di funzioni* hash H_0, H_1, ..., H_I, ... che soddisfino le seguenti condizioni:

- **Range condition**: la funzione H_I deve essere a valori in $[0, 2^I \cdot P_0 - 1]$, cioè la funzione H_I possa generare tutti gli indirizzi
- **Split condition:** Per ogni $I > 0$, per ogni chiave k, e per ogni valore di $H_I(k)$ deve valere la seguente relazione:

$$H_I(k) = H_{I-1}(k) \, oppure \, H_I(k) = H_{I-1}(k) + 2^{I-1} \cdot P_0$$

ovvero lo split di un bucket deve lasciare una chiave o nel bucket stesso o allocarla nel bucket *buddy*.

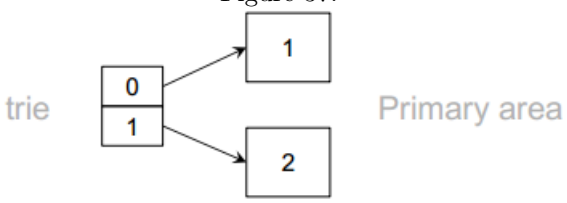
La famiglia di funzioni $H_I(k) = k\%(2^I \cdot P_0)$ soddisfa entrambe le condizioni

Dynamic hashing A differenza del *virtual hashing* il *dynamic hashing* non raddoppia l'area primaria, bensì la fa cresce di un bucket alla volta, utilizzando la struttura ausiliaria della *directory* come un albero (*trie*) binario. Il *dynamic hashing* (Larson '78) evita dunque di ricorrere a tecniche di raddoppio (che oltre a causare appesantimenti non trascurabili nel momento in cui l'area primaria viene raddoppiata, riducono il fattore di utilizzazione) facendo uso di una struttura ausiliaria, la *directory*, organizzata come un *trie albero*. L'idea di base (comune all'*extendible hashing*) è quella di far uso di una funzione hash che, dato il valore k, restituisce una *pseudo-chiave binaria* $H(k) = b_0, b_1, b_2, \ldots$ (cioè la chiave k viene utilizzata come seme di un generatore di numeri pseudo-casuale). La situazione ideale è quella in cui l'insieme di pseudo-chiavi da gestire è tale per cui la probabilità di generare un bit (0 o 1) è la stessa, cioè $Pr\{b_i = 1\} = \frac{1}{2}$ ovvero vi sia una ripartizione bilanciata per ogni posizione considerata.

Vediamo dunque l'uso dell'albero *trie*. Il *trie* serve ad organizzare la ricerca ed ogni foglia indirizza i bucket dell'area dati. Per cercare (o inserire) un avolre di chiave *si segue, fino ad una foglia, il cammino del trie corrispondente alla pseudo chiave*, cioè viene utilizzata la funzione Hash per generare la pseudo-chiave che mi porterà al bucket in cui la chiave dovrebbe essere presente.

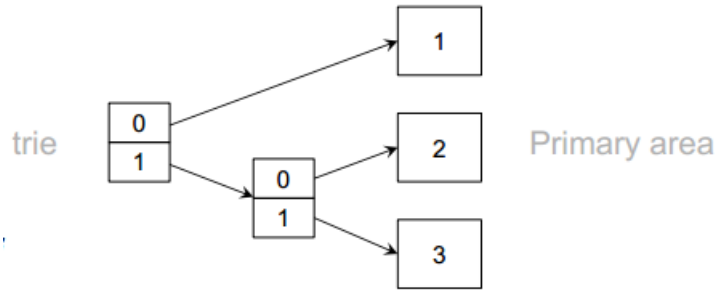
Nell'esempio in figura (Fig. 37) il bucket 1 contiene tutti i valori di chiave la cui pseudo-chiave è del tipo $0\ldots$ (cioè che iniziano per 0) mentre il bucket 2 quelli la cui pseudo-chiave è del tipo $1\ldots$ (cioè che iniziano per 1).

Figure 37:



Bisogna ovviamente gestire il caso di *overflow* di un bucket. L'espanzione dell'area primaria avviene aggiungendo *un bucket alla volta*, *ridistribuendo* i record tra il bucket saturo e il suo *buddy* e *aggiungendo* un nodo al *trie*. L'esempio in figura (Fig.) meglio potrà chiarire. Il bucket 2 dell'esempio precedente (Fig. 37) è in overflow e deve subire il processo di split. La nuova situazione prevede che il bucket 2 ora contenga tutti i valori di chiave la cui pseudo-chiave è del tipo $10\ldots$ mentre il bucket 3 quelli la cui pseudo chiave è $11\ldots$ (mentre il bucket 1 è rimasto invariato e conterrà le chiavi il cui pseudo codice inizia per $0\ldots$).

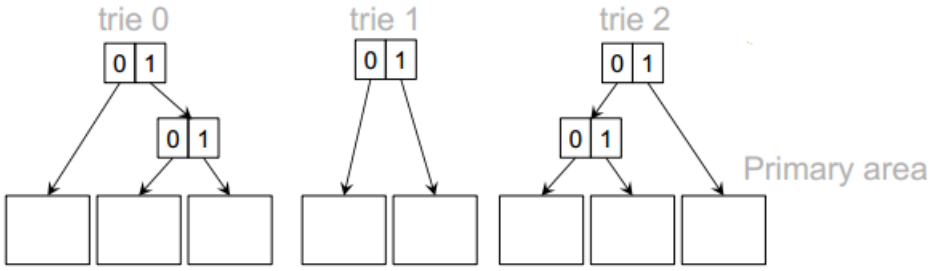
Figure 38:



Si notri che se il *trie* è in memoria centrale basta un singolo accesso per recuperare un record, invece se il *trie* non risulta stare in memoria centrale le prestazioni dipendono dal bilanciamento dell'albero stesso in termini di numero di nodi indice da dover recuperare. Le prestazioni nel caso peggiore non sono buone, infatti a seconda dell'insieme di pseudo-chiavi l'inserimento di un record può comportare più di uno split. Se dopo una cancellazione in un bucket il numero di recordo contentui nel bucket e nel suo buddy diventa minore o uguale alla capacità C, i bucket vengono fusi e si elimina una foglia dal *trie*. Si noti che in media il fattore di utilizzazione dei bucket è di circa il 70%.

Esiste anche una variante al *dynamic hashing*: invece di avere un solo grande albero *trie*, vengono utilizzati più trie. Inizilamente si allocano un numero di bucket pari a P. Viene creata una funzione hash statica H_0 e a seguito di un *overflow* vengono generati P *trie* le cui radici sono indirizzate da H_0. In pratica la funzione H_0 fa da vera funzione hash indirizzando la chiave ad un trie, una volta giunti nel trie si utilizza una nuova funzione hash che genera la pseudo-chiave cadendo nel caso precedentemente studiato. Si veda l'esempio in figura (Fig. 39) in cui $H_0(k) = k\%3$.

Figure 39:

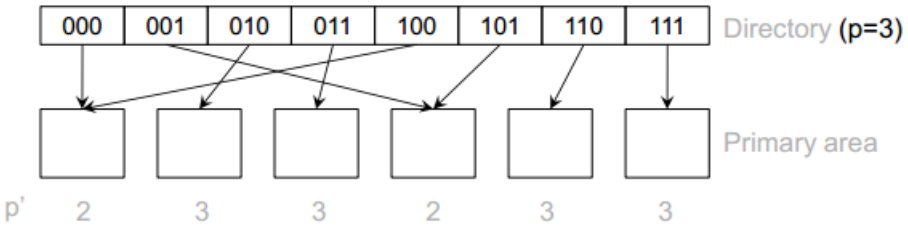


Extendible hashing Molto simile al *dynamic hashing*, da cui si differenzia per la gestione della *directory,* l'*extandible hashing* (Fagin, '79) ha la peculiarità

di garantire non più di due accessi I/O. La *directory* è formata da 2^p celle di indirizzi compresi tra $[0, 2^p - 1]$. Il valore $p \geq 0$ è detto *profondità della directory*. Una funzione *hash* associa ad ogni chiave k_i una **pseudo-chiave binaria** (in sostanza k viene usato come seme di un Pseudo-Random Number Generator), cioè la funzione hash produce una sequenza binaria lunga almeno p *bit* $H(k_i) = \ldots, b_2, b_1, b_0$. Di questa sequenza si considerano i p *bit meno significativi*: essi verranno utilizzati per accedere direttamente a una delle 2^p celle della *directory* stessa, ognuna contenente un puntatore a un bucket. Il numero dei bucket è sempre minore o uguale al numero di celle: se essi coincidono allora ogni cella della directory punta ad un bucket diverso, se minore indica che alcune celle puntano allo stesso bucket.

Ogni bucket ha una *profondità locale p'* con $p' \leq p$ (il valore di *p'* viene mantenuto nel bucket) che indica *il numero effettivo di bit* usati per allocare le chiavi nel bucket stesso. L'esempio (Fig. 40) può aiutare nella comprensione: il bucket contenente la chiave 258 (la cui funzione *hash* produce una sequenza binaria $\ldots 001$) ha profondità locale $p' = 2$, di conseguenza lo stesso bucket conterrà le chiavi le cui pseudo-chiavi (il risultato della funzione hash) risultino essere sia $\ldots 001$ sia $\ldots 101$ (poichè bastano i soli 2 bit meno significativi 01 per allocare le chiavi nel bucket).

Figure 40:



Split di un bucket Cosa succede se un bucket a profondità locale p' risulta essere in *overflow*?

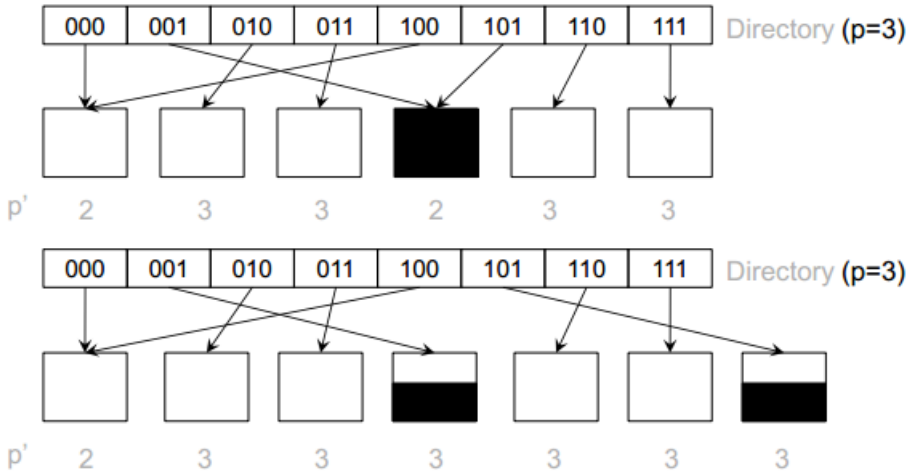
Bisogna distinguere due casi:

- $p' < p$
- $p' = p$

Nel primo caso (p'<p) viene generato (si alloca) un *nuovo bucket* e si distribuiscono le chiavi tra i due bucket utilizzando il bit *p'+1*-esimo aggiuntivo della pseudo-chiave (in quanto prima era un 'bit' superfluo perchè più celle puntavano allo stesso bucket, ora invece questo bit in più permette l'identificazione del bucket appena creato da quello originario). Per i due bucket si modifica la loro *profondità locale* incrementandola di 1. Inoltre proprio perchè *p'<p* esiste sicuramente una cella che può indirizzare il nuovo bucket (poichè c'erano più celle che puntavano allo stesso bucket), pertanto si modifica il puntatore di

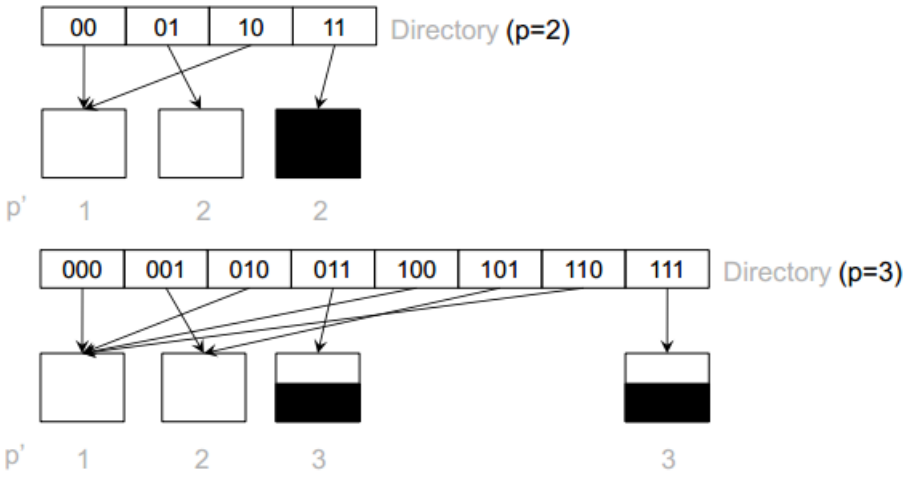
questa cella al bucket appena generato. Si veda l'esempio in figura (Fig. 41) per una maggiore comprensione.

Figure 41:



Il secondo caso ($p'=p$) richiede alcuni accorgimenti ulteriori. Poichè $p'=p$ vuol dire che non esiste alcuna cella che possa indirizzare il nuovo bucket (poichè il bucket originario è indirizzato da una sola cella) eventualmente ottenuto dallo split. Per risolvere il problema si *raddoppia* la *directory* e si incrementa il valore della profondità della directory p di 1, si copiano dunque i valori dei puntatori nelle nuove celle corrispondenti (avendo raddoppiato la directory son passato da 2^p celle a 2^{p+1}, e le nuove 2^p celle dovranno puntare ai bucket esistenti). Infine si esegue lo *split* ricadendo nel caso precedente di $p'<p$. Si veda l'esempio in figura (Fig. 42) per una maggiore comprensione.

Figure 42:

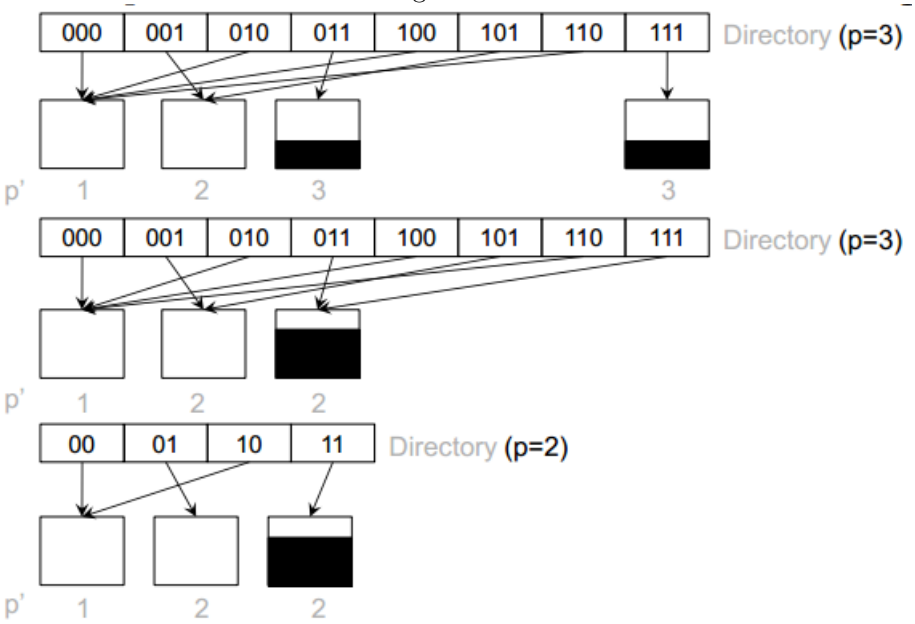


Rimane un problema irrisolto che andremo a illustrare attraverso l'esempio. Si provi ad immaginare che il bucket in *overflow* puntato dalla cella con indizzo 11 abbia al suo interno solo record le cui pseudo-chiavi terminino tutte per 111. Una volta raddoppiata la *directory* tutti questi record si porteranno nel bucket indirizzato dalla cella contenente 111, portando ad avere il bucket ancora in *overflow* mentre il suo compagno 011 risulterebbe vuoto. A questo punto basta iterare il procedimento, esso dovrò convergere in quanto valutando sempre un bit in più della pseudochiave ci ritroveremo a diversificare i vari record e a rendere il bucket non più in overflow. Questo vuol dire che una volta raddoppiata la *directory* bisogna controllare che i bucket non siano ancora in overflow e in caso rieseguire il procedimento.

Cancellazione Anche in caso di cancellazione di un record bisogna procedere con cautela. Infatti, se si cancella un record in un bucket con profondità locale *p'* e la somma dei record contenuti nel bucket e nel bucket *buddy* (compagno) associato diventa $\leq C$ (Capacità del bucket), i due bucket vengono fusi e il *bucket* risultante avrà profondità locale *p'-1*. Il bucket buddy di profondità locale p' di un determinato bucket di profondità locale p' (hanno stessa profondità) è quello che è identificato da una sequenza di *p'* bit di cui solo il bit più significativo (quello più a sinistra) risulta diverso (ad esempio bucket compagni sono (101,001), (111,011), (000,100), ...).

Se si fondono gli unici due bucket che hanno profondità locale $p' = p$ allora è possibile *contrarre* (*dimezzare*) *la directory*. Poichè per verificare che non esistono più bucket a profondità p richiede, nel caso peggiore, di esaminare metà dei bucket, di conseguenza è conveniente utilizzare una tabella delle profondità locali che per ogni valore $p' \leq p$ mantiene il numero $P(p')$ di bucket a profondità *p'*. Si veda l'esempio in figura (Fig. 43) per una maggiore comprensione.

Figure 43:



Possiamo fare delle considerazioni finali: ogni raddoppio coinvolge tutta la *directory*, si potrebbero avere dunque problemi in caso di concorrenza all'accesso all'indice. Una possibile soluzione prevede l'uso di una *directory* a più livelli e conseguentemente l'indice non risulta più necessariamente binario.

Linear Hashing È la prima tecnica che guardiamo che non fa uso della *directory*. Similmente all'*extendible hashing*, il *linear hashing* espande la memoria linearmente, cioè aggiungendo un bucket alla volta, utilizzando ancora una volta la tecnica dello *split*. A differenza però delle tecniche con *directory* bisogna ora utilizzare una tecnica più "semplice" per scegliere quale bucket andare a dividere. In breve il *linear hashing* è una tecnica ad *espansione lineare* che non essegue lo split del bucket in *overflow* ma bensì di un altro bucket scelto secondo un criterio specifico: ciò rende non necessaria una *directory*, occorrerà ovviamente gestire ancora l'*overflow* (siccome lo split del bucket non riguarda il bucket in overflow, bisognerà gestire l'*overflow* con le tecniche inizialmente descritte: *open addressing* o *metodi a liste*) e infine l'area primaria crescerà "linearmente". La semplice tecnica utilizzata per decidere quale bucket dividere è la seguente: *si divide il bucket successivo all'ultimo bucket diviso.* Si noti che se il bucket su cui si esegue lo *split* era vuoto abbiamo diviso un bucket che non era necessario dividere: l'idea del *linear hashing* è di pensare che prima o poi, continuando ad eseguire lo split ogni volta che c'è overflow, arriverò a suddividere proprio il bucket in *overflow*.

Indicato con P_0 il numero dei bucket inizialmente allocati, si utilizza come funzione hash $H_0(k_i) = k_i\%P_0$. Si mantiene anche un puntatore **SP** (Split Pointer) al prossimo bucket che dovrà essere suddiviso (inizialmente dunque $SP = 0$). Se si verifica un *overflow* si devono eseguire le seguenti operazioni:

1. Si aggiunge in coda un bucket di indirizzo $P_0 + SP$
2. Si riallocano i record del bucket SP (quello che è stato diviso), inclusi quelli eventualmenti presenti in area di overflow ad esso associati, utilizzando però una nuova funzione hash $H_1(k) = k\%(2P_0)$ che suppone che la nostra area primaria sia raddoppiata. L'utilizzo temporaneo di questa nuova funzione hash è necessario per riuscire a raggiungere anche il nuovo bucket creato all'indirizzo $P_0 + SP$.
3. Si incrementa SP di 1.

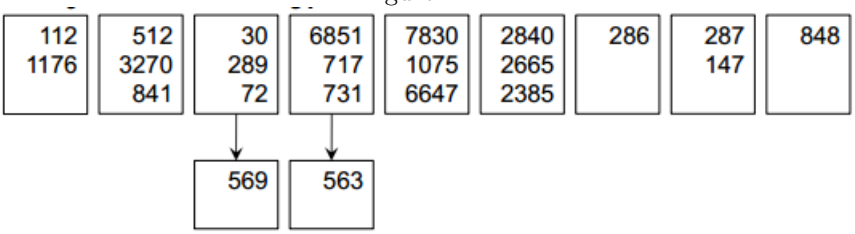
La cosa interessante riguarda l'utilizzo delle due funzioni: infatti se H_0 dava come risultato *SP*, allora H_1 può dare come risultato solo o *SP* o P_0+*SP*. Un esempio potrà chiarire: si supponga inizialmente $P_0 = 5$, e si supponga che una chiave k_i dia come risultato da $H_0(k_i) = k_i\%5 = 2$, questo implica che la chiave k_i doveva avere per forza come ultima cifra o 2 o 7, se applichiamo la stessa chiave a $H_1(k_i) = k_i\%(2 \cdot 5) = k_i\%10 \rightarrow$ allora essa potrà dare come risultato solo o 2 o 7, a seconda di quale fosse l'ultima cifra di k_i.

Si noti che dopo aver effettuato esattamente P_0 split (corrispondendi a P_0 overflow), l'area di memoria è raddoppiata, in quanto il numero di bucket è ora pari a $2P_0$: bisogna dunque predisporre i parametri ad una nuova espansione ponendo $SP = 0$ (SP deve tornare all'inizio), imponendo $H_0(k) = H_1(k)$ (cioè modifico la funzione hash originale) e infine modificando anche $H_1(k) = k\%(2^2 \cdot P_0)$. In generale quando bisogna predisporsi alla j-esima espansione le funzioni due funzioni hash H_0 e H_1 da utilizzare diventano $H_0(k) = k\%(2^{j-1} \cdot P_0)$ e $H_1(k) = k\%(2^j \cdot P_0)$.

Supponiamo ora di avere una chiave k_i da dover inserire, come faccio a sapere quale funzione hash utilizzare? Il problema sorge poichè se utilizzassi sempre la funzione H_0 avrei reso inutile l'allocazione del nuovo bucket all'indirizzo SP+P_0. Il problema si risolve in maniera molto semplice: se $H_0(k_i) \geq SP$ allora effettivamente posso utilizzare la funzione H_0 per allocare la chiave k_i al bucket indicato dal risultato della funzione H_0, altrimenti sono costretto a calcolare $H_1(k)$ e utilizzare il risultato di questa seconda funzione hash.

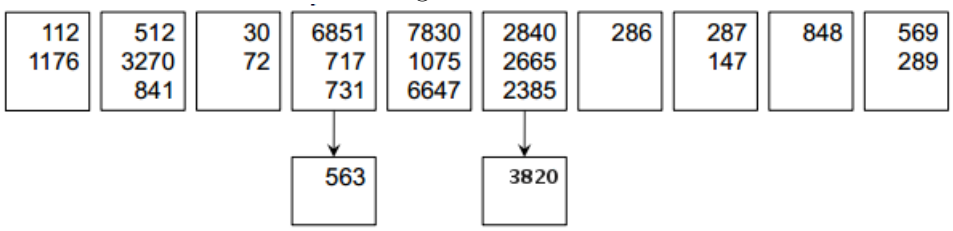
Introduciamo un esempio (Fig. 44 e Fig. 45) per maggiori chiarimenti.
Si immagini $P_0 = 7$, $C = 3$, $C_{OV} = 2$ e $SP = 2$ (SP=2 vuol dire che ci sono stati già due *overflow* che ha portato lo split del bucket 0 e del bucket 1).

Figure 44:



Inserendo ora la chiave $k_i = 3820 \rightarrow H_0(3820) = 3820\%7 = 5$ si genera un nuovo *overflow* nel bucket 5 (poichè pieno). Questo comporta ad allocare un nuovo *bucket* in posizione 7+2=9, a ridistribuire le chiavi presenti nel bucket SP=2 cioè del terzo bucket (siamo in 0-based) e ad incrementare SP di uno risultando SP=3. Si noti come nella nuova ridistribuzione totale (Fig. 45) il bucket 5 andato in *overflow* ha dovuto gestire in modo autonomo il proprio *overflow*, utilizzando, nell'esempio, un bucket in area di overflow.

Figure 45:

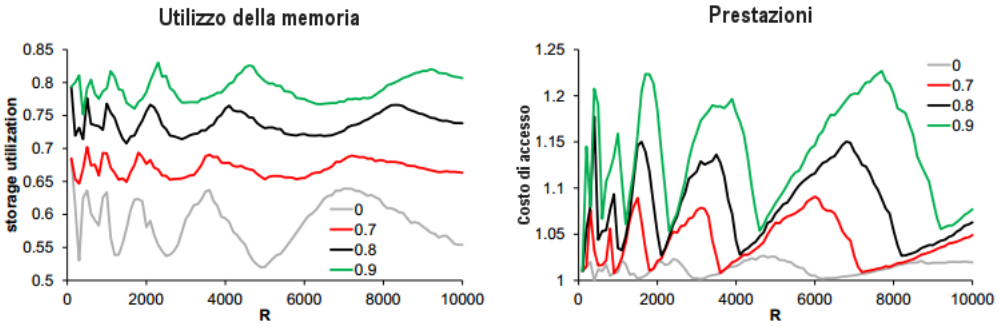


L'allocazione del bucket 9 ha causato l'eliminazione di un bucket di *overflow*. Il prossimo *overflow* causerà lo split del bucket 3 (cioè il quarto). Dopo 7 split l'area primaria è raddoppiata e si impostano i valori di SP e delle funzioni hash in modo da essere pronti ad una nuova espansione.

L'assenza di una *directory* e la politica semplice degli split rendono di facile realizzazione l'intera struttura, inoltre la gestione dell'area primaria (sia per espansione che contrazione) è immediata, in quanto i bucket vengono sempre aggiunti (e rimossi) in coda. Di contro però l'utilizzazione della memoria allocata è decisamente bassa (varia tra 0.5 e 0.7), inoltre la gestione dell'area di *overflow* presenta problemi simili a quelli visti nella gestione dell'hashing statico ed infine le catene di *overflow* relative ai buket di indirizzo molto alto, non ancora suddivise, possono diventare molto lunghe.

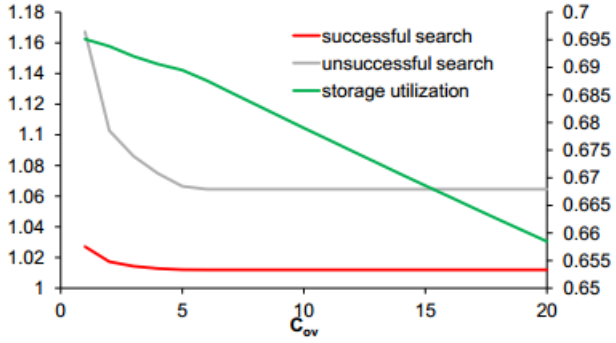
Esiste una variante al modello presentato che prevede di non effettuare lo *split* di un bucket se questo non ha raggiunto un livello di utilizzazione minimo u_{min}, tutto a beneficio di una migliore utilizzazione della memoria. All'aumentare dell'utilizzazione di memoria auemntano anche i costi di ricerca in quanto aumentano i record in *overflow*. I minimi dei costi di accesso corrispondono ai massimi dell'utilizzazione di memoria.

Figure 46:



Infine all'aumentare la capacità dei bucket in overflow C_{OV} si riduce, ovviamente, la lunghezza delle catene di *overflow*, di conseguenza si riduce il costo della ricerca. Oltre ad un certo limite però, si può notare che aumentando ulteriormente C_{OV} si ha solamente uno spreco di memoria.

Figure 47:



Linear Hashing Ricorsivo È una implementazione del *linear hashing* anche per gestire l'area di *overflow*: vengono utilizzate strutture *linear hashing* ognuna per ogni livello, cioè una per l'area *principale*, una per il primo livello di area di overflow, una per il secondo livello di area di overflow, etc...

Spiral Hashing Il problema del *linear hashing* è quello di andare a dividere un *bucket* che non ha problemi di *overflow*: sarebbe bello dunque applicare una regola semplice come quella usata dal *linear hashing* però al bucket che effettivamente va in *overflow*. Lo *spiral hashing* utilizzando questa idea, tenta di avvicinarsi ad un ottimo connubio. Utilizzando una funzione *hash* non *equidistributiva*, quindi con probabilità di allocare i record su di un bucket piuttosto che

su di un altro, rende i *bucket* non equiprobabili all'*overflow*: nella specifica lo *spiral hashing* tenta di rendere fortemente probabile all'*overflow* il primo bucket attraverso la funzione *hash* e conseguentemente di applicare lo split (qui la regola semplice sullo split) sempre al *primo* bucket. L'organizzazione deve il suo nome al fatto che lo spazio di memoria viene pensato come una spirale piuttosto che a una retta, in cui l'area primaria viene immaginata come una rivoluzione della spirale stessa, univocamente definita da un angolo z che la sottende. Dati molti vantaggi, lo *spiral hashing* richiede però molta cura nell'essere costruito: infatti fra i primi problemi c'è il disaccoppiamento tra gli indirizzi fisici e gli indirizzi logici dei bucket.