ARGOMENTO 10

Indicizzazione di file & costi

Indicizzazione dei file

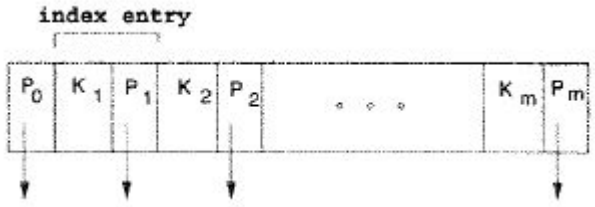
- Un altro modo per rendere più efficienti le operazioni sul file dei dati è quello di creare degli indici esterni, indipendentemente dal fatto che il file sia ordinato o meno
  - Il *data file* è il file che contiene i dati veri e propri
  - L'*index file* è il file che contiene l'indice dei dati (può esistere più di un *index file*)
- Noi vedremo due diversi tipi di indice (quelli di gran lunga più usati dai più comuni DBMS)
  - B+ tree
  - Hash

Indici primari, secondari, unici

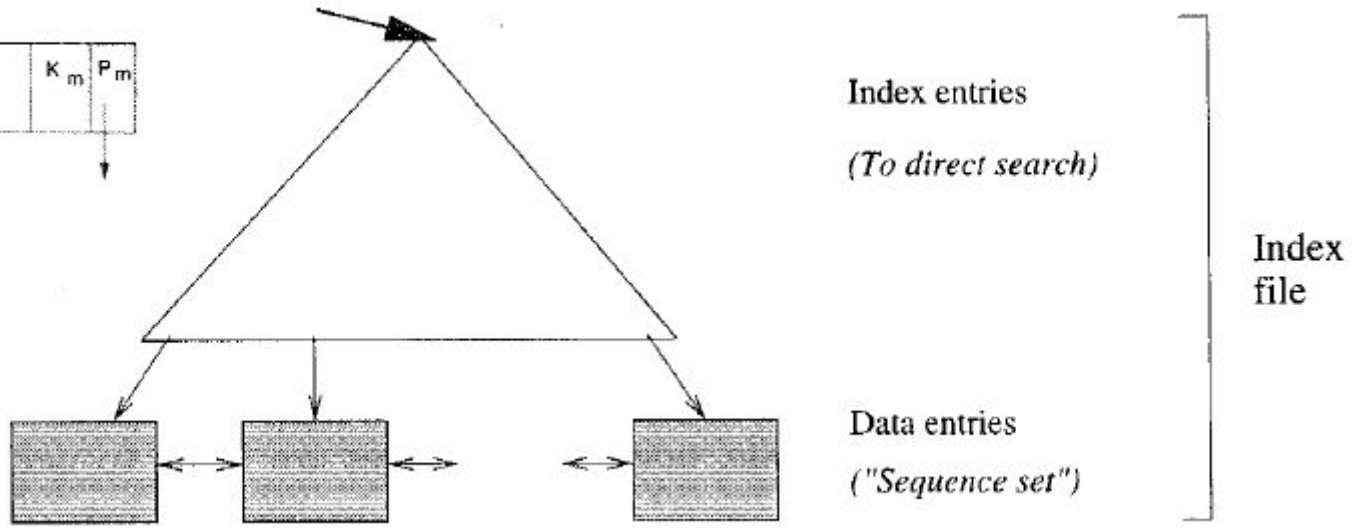
- Un indice si dice:
  - **Primario** (*primary)* se contiene la chiave primaria della relazione. Non ci sono duplicati nelle entries dell'indice
  - **Secondario** (*secondary*) se la chiave di ricerca è diversa dalla chiave primaria. Può contenere duplicati nei valori della chiave di ricerca
  - **Unico** (*unique*) se la chiave di ricerca è una chiave candidata. Anche in questo caso non ci sono duplicati

B+Trees

- Si tratta di un tipo di indice **dinamico** ad albero, dove i puntatori ai dati sono memorizzati solo nei nodi foglia
  - I nodi interni dell'albero contengono *index entries* e servono a guidare la ricerca
  - I nodi foglia contengono i puntatori ai dati del *data file*
  - I nodi foglia sono collegati tra di loro (*sequence set*)



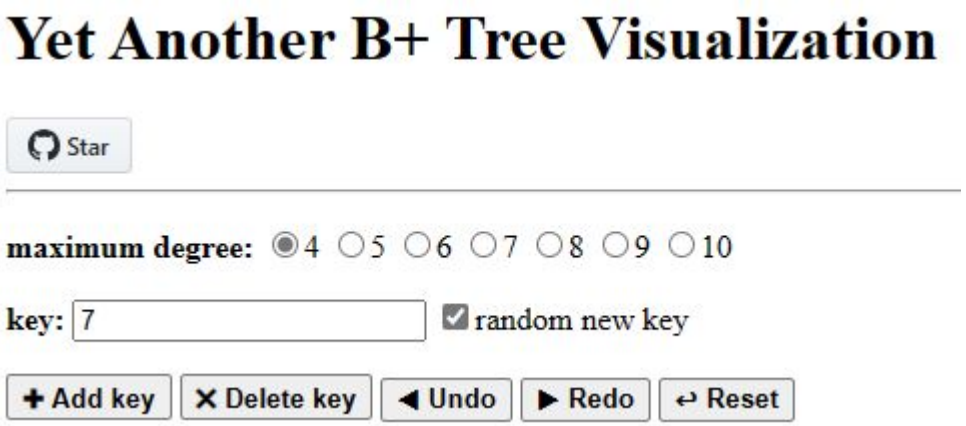
Formato di una *index page*



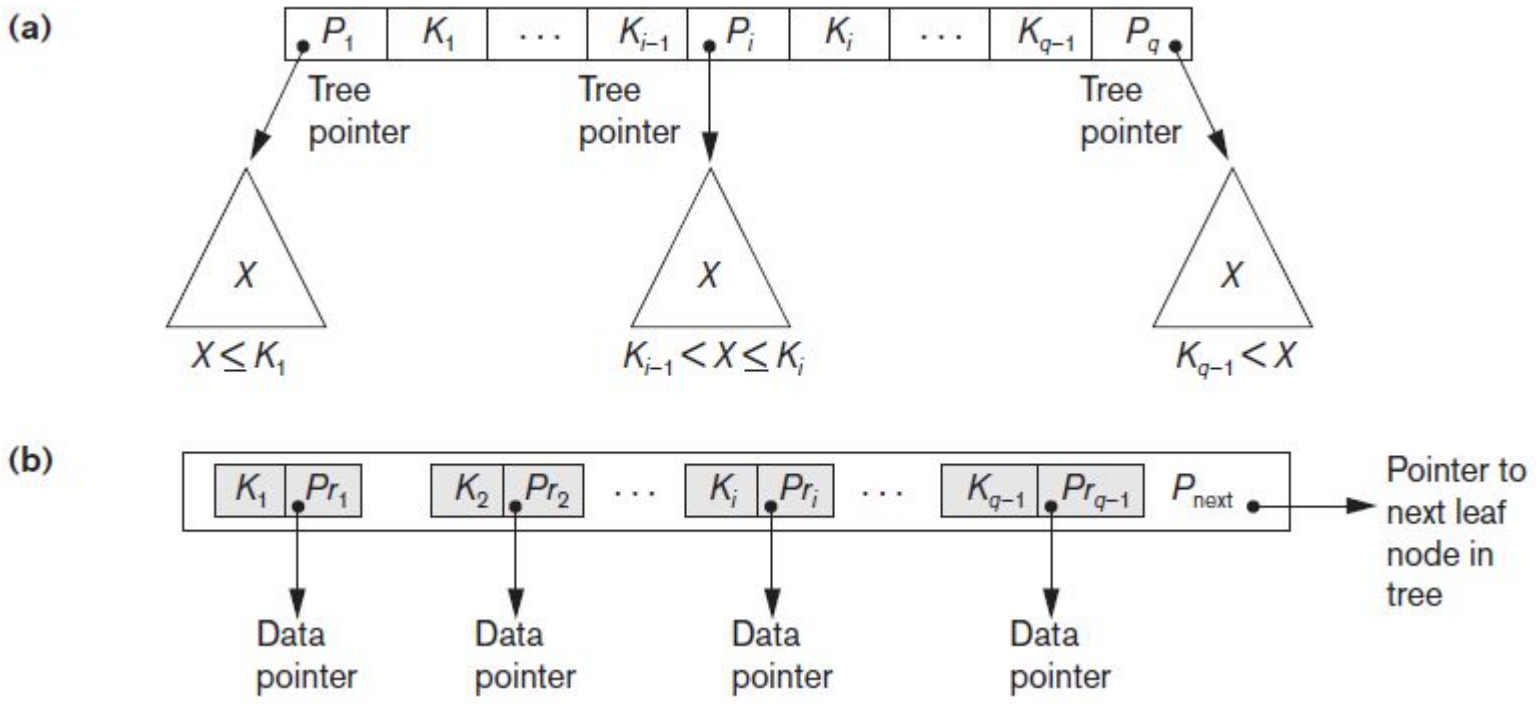
B+ -Trees: costruzione

- Usiamo un simulatore per capire come si costruisce in indice di tipo B+ Tree:

https://roy2220.github.io/bptree/visualization/#4



B+ -Trees



(a) Nodi Interni di un B+-tree con *q−1* valori di ricerca (b) Nodi foglia di un B+-tree con *q−1* valori di ricerca *e q−1* puntatori ai dati

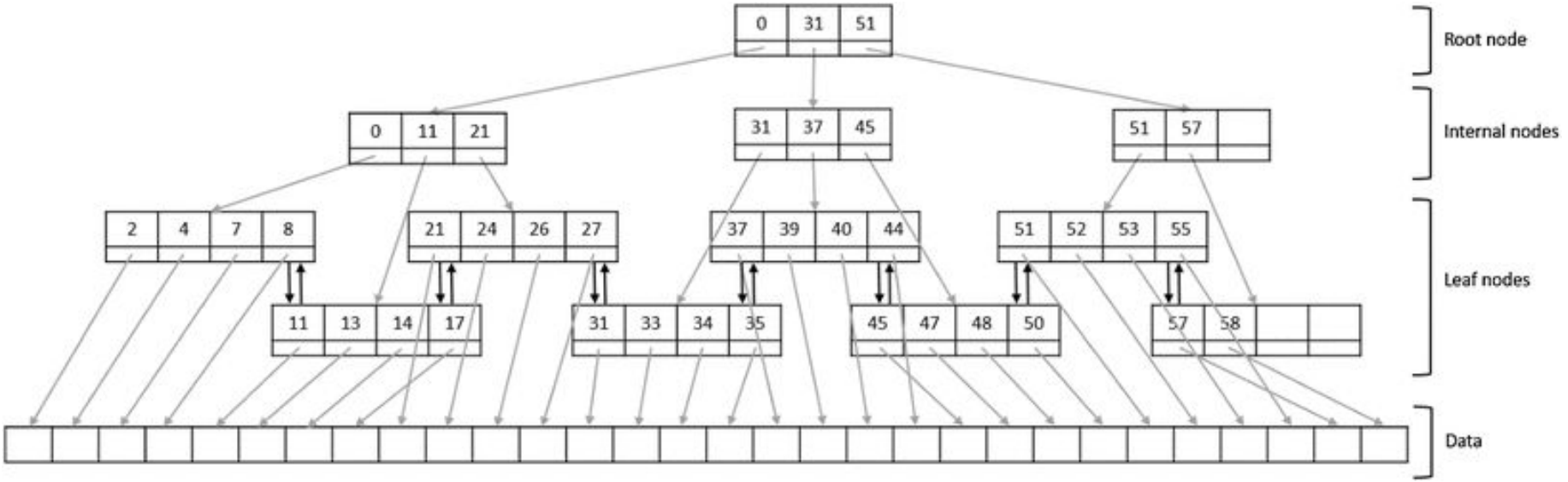
B+Trees

- Le proprietà principali di un B+ Tree sono:
  - Le operazioni di inserimento e cancellazione nell'albero lo mantengono bilanciato (cioè, tutte le foglie sono alla stessa profondità)
  - È garantita un'occupazione almeno al 50% di ogni nodo dell'albero (ad eccezione della radice)
  - La ricerca di un record richiede soltanto di discendere l'albero fino all'appropriato nodo foglia
  - Siccome l'albero è bilanciato, possiamo definire come **altezza dell'albero** la lunghezza di qualunque percorso dalla radice alle foglie (0 per il nodo radice)
  - Grazie alle sue proprietà, raramente l'altezza di un B+Tree supera il valore di 3 o 4

B+tree : il costo del *lookup*

- Per costo di *lookup* si intende il numero di pagine che dobbiamo aprire per trovare il nodo foglia che contiene i puntatori ai dati che stiamo cercando
- Sia B il numero massimo di figli che può avere un nodo interno di un B+tree, *n* il numero di nodi foglia e *A* la chiave di ricerca
- Il costo sarà $\lceil \log_B(|R.A|) \rceil$ e a ogni passo divido per B il numero di nodi rimanenti (*n* all'inizio, *n / B* al secondo passo e così via).

B+tree : il costo del *lookup*



B+tree : EQUALITY SEARCH - *Unclustered files*

- Sia la relazione R memorizzata su un file non ordinato (*unclustered*)
- Immaginiamo di voler trovare tutte le tuple di una relazione R che corrispondono a un filtro:

```
SELECT * FROM R WHERE age = 30;
```

- Dato B il numero massimo di figli che può avere un nodo, il costo della ricerca sarà:

$$\text{Costo}_{\text{EqSearch}} = \lceil \log_B(|R.\text{age}|) \rceil + |R_{\text{age}=30}|$$

B+tree : EQUALITY SEARCH - *Clustered files*

- Sia R memorizzata su un file ordinato (*clustered*) sulla chiave di ricerca (*age*)
- Immaginiamo di voler trovare tutte le tuple di una relazione R che corrispondono a un filtro:

```
SELECT * FROM R WHERE age = 30;
```

- Dato B il numero massimo di figli che può avere un nodo:

$$\text{Costo}_{\text{EqSearch}} = \lceil \log_B(|R.\text{age}|) \rceil + \left\lceil \frac{|R_{\text{age}=30}|}{\left\lfloor \frac{P}{t_R} \right\rfloor} \right\rceil$$

B+tree : Ricerca per intervalli

- Immaginiamo di voler trovare tutte le tuple di una relazione R che corrispondono a un filtro:

```
SELECT * FROM R WHERE age > 18 AND < 30;
```

- Una volta letta la foglia che contiene tuple con age = 18, possiamo usare i puntatori ”orizzontali” tra le foglie per trovare tutte le pagine del data file che ci servono

B+tree vs. sorted file

- La ricerca su B+tree è in generale più efficiente di quella su sorted file perché:
  - Non ci sono i costi per mantenere ordinato il file dei dati
  - la ricerca è logaritmica in base B (e non in base 2).
- Esempio: P_R = 1.000.000 / 100 = 10.000
- $\log_2(10000) = 13$
- Con $B = 100$ e $|R.A| = 400$, $log_B|R.A| = 1{,}3$
- Se non è noto il valore di B e di $|R.A|$, in genere si assume che il costo di lookup sia 3

B+tree : INSERT

- L'inserimento di una nuova tupla ha lo stesso costo dell'Equality Search, più il costo di creare (eventualmente) una nuova pagina e di salvare la pagina modificata su disco:

$$\text{Costo}_{\text{Insert}} = \lceil \log_B(|R.\text{age}|) \rceil + 1 + 1$$

- Anche qui, non stiamo considerando il costo di aggiornare l'indice dopo una INSERT.

B+tree : SCAN

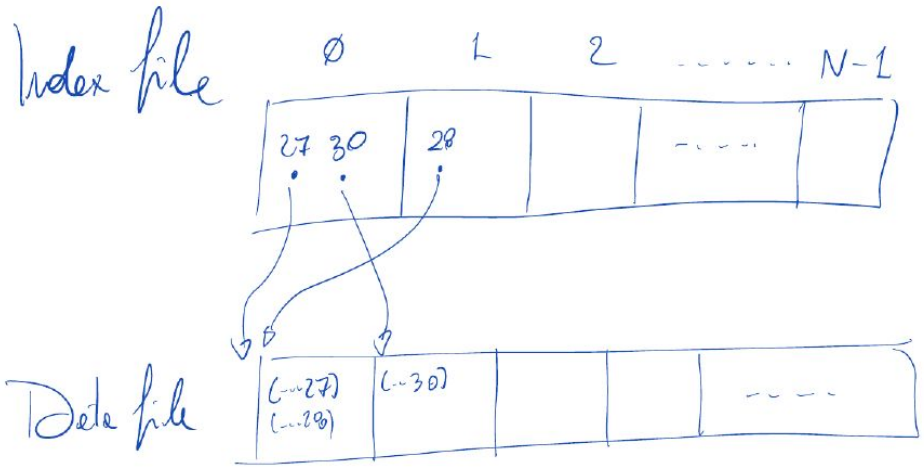
- Lo SCAN ha il solito costo di aprire tutte le pagine in cui è memorizzata la relazione R:

$$\text{Costo}_{\text{Scan}} = P_R$$

Hash Indexes: l'idea

- Si sceglie una funzione di Hash (ad esempio, per i numeri interi, $h(x) = x\%N$, dove N è l'operatore resto della divisione)
- Si allocano i diversi valori di un certo attributo in un *bucket* («secchiello») in base al valore della funzione di hash prescelta
- Se la pagina del bucket contiene troppi puntatori, il DBMS dovrà creare una pagina di overflow, che sarà connessa alla pagina principale del bucket e che potrà contenere i nuovi puntatori al file di dati

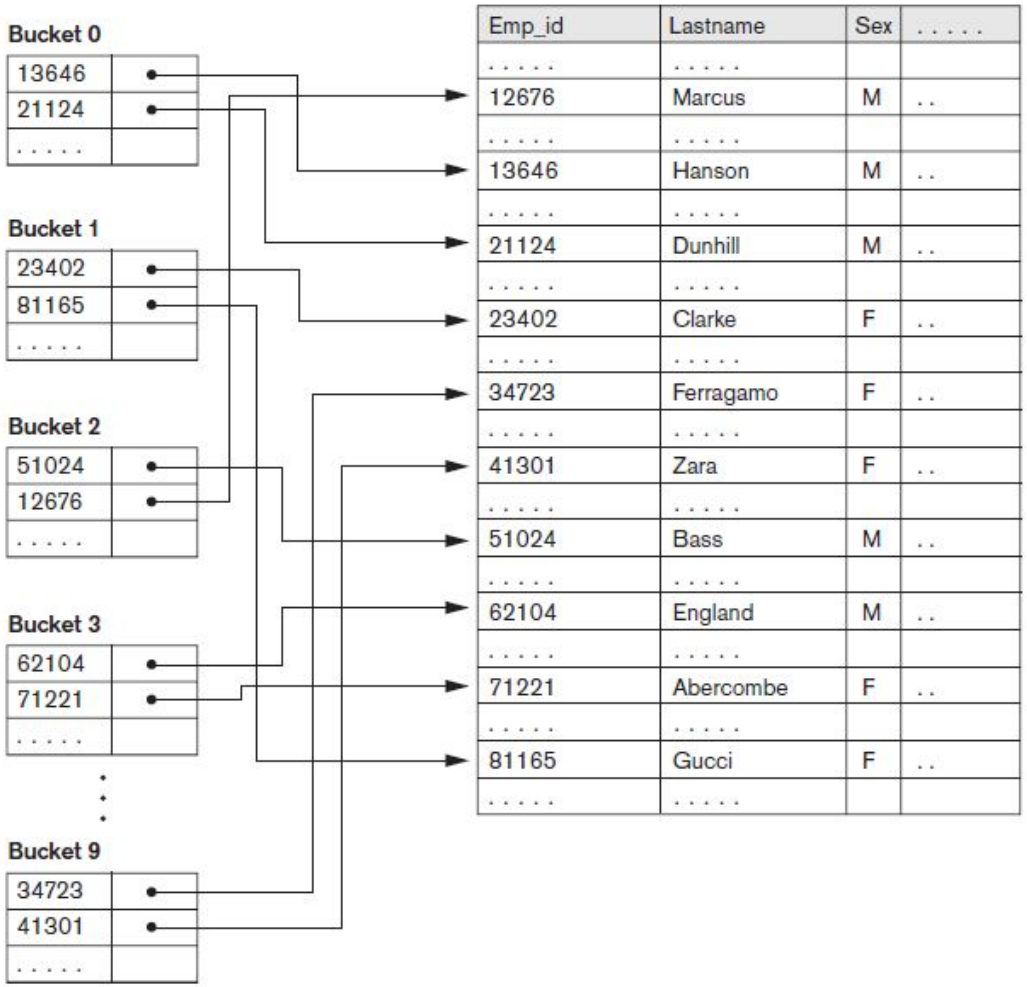
Hash Indexes: esempio



Esempio per l'attributo *age*

- Per trovare una tupla con un certo valore di *age x* basterà applicare la funzione di hash *h* a *x*
- Questo ci darà il bucket che contiene il puntatore alla pagina del data file con quella tupla (al costo di leggere la o le pagine che contengono il bucket nell'index file)

Hash Indexes (cont'd.)



Hash Index: EQUALITY SEARCH

- Immaginiamo di voler trovare tutte le tuple di una relazione R che corrispondono al filtro:

```
SELECT * FROM R WHERE age = 30;
```

- Il costo sarà:

$$\text{Costo}_{\text{EqSearch}} = L_h + |R_{A=c}|$$

- dove:
  - L_h è il costo del lookup nel file di indice
  - $|R_{A=c}|$ è il numero di tuple che hanno valore *c* per l'attributo *A* nella relazione *R*.

Hash Index: EQUALITY SEARCH

- Il costo del lookup nel file di indice (L_h) sarà dato dal calcolo della funzione *h* sul valore cercato (che si fa in memoria centrale a costo 0) e dalla lettura della pagina che contiene il *bucket*
- *Se* ci sono pagine di overflow, è necessario leggerle tutte, anche se in generale – se la funzione di hash è scelta bene – non ce ne saranno molte per default si assume un valore medio di 1,2
- L'hash index è quindi MOLTO efficiente in caso di *Equality Search*

Hash Index: ricerca su intervalli

- Per la ricerca su intervalli, invece, l'hash index non è efficiente, perché in generale non è detto che tuple con valori vicini di un attributo siano nello stesso bucket dell'index file
- Esempio: supponiamo di voler trovare tutte le tuple di una relazione R che corrispondono al filtro:

```
SELECT * FROM R WHERE age > 18 AND < 30;
```

- Per ogni valore dell'intervallo, sarà necessario rifare il *lookup*, trovare il bucket giusto nell'*index file* e poi leggere tutte le pagine puntate da quel bucket
- In generale, un *hash index* non permette quindi di leggere le tuple in ordine rispetto alla chiave di ricerca.

Hash Index: INSERT

- L'inserimento di una nuova tupla richiede:
  - di aprire l'ultima pagina del file di dati e appendere la nuova tupla in fondo (costo: 1+1)
  - Eseguire il lookup per trovare il bucket in cui inserire il puntatore al nuovo valore (costo: L_h)

$$\text{Costo}_{\text{Insert}} = L_h + 1 + 1$$

- Anche in questo caso ignoreremo il costo di aggiornare l'indice dopo una INSERT.

Hash Index: DELETE

- La cancellazione delle tuple in base al valore *c* di un attributo *A* richiede:
  - eseguire il lookup per trovare il bucket con i puntatori alle tuple con valore *c* (costo: L_h)
  - aprire una alla volta le pagine del file di dati che contengono tuple con valore *c* per l'attributo *A*, rimuovere le tuple e riscrivere la pagina modificata (costo: $2 * |R_{A=c}|$)
- Il costo totale sarà quindi $L_h + 2 * |R_{A=c}|$

Hash Index: SCAN

- Lo SCAN ha il solito costo di aprire tutte le pagine in cui è memorizzata la relazione R:

$$\text{Costo}_{\text{Scan}} = P_R$$

- Possiamo quindi ignorare l'*index file* e accedere direttamente al file dei dati

Hash Index: considerazioni generali

- L'Hash index è molto efficiente, tranne che per la ricerca per intervallo
- Tuttavia, osserviamo che, prima o poi, i bucket richiederanno sempre più pagine di overflow e quindi il costo di *lookup* aumenterà
- Il DBMS dovrà periodicamente riorganizzare l'*index file*, utilizzando una funzione di hash con più bucket
- Questo però richiederà di spostare i vari puntatori nei nuovi bucket, con relativo costo
- Questo approccio si chiama **Extendible Hash Index**

Index matching

- Finora, abbiamo considerato solo casi in cui la condizione di selezione (nella clausola WHERE) era costituita da un singolo attributo (es. city = ‘Trento’)
- Come vanno usati gli indici nel caso in cui ci siamo molteplici condizioni di selezione? Per esempio:
  - city = ‘Trento’ AND age = 30
  - city = ‘Trento’ OR age = 30
- A questo scopo, introduciamo il concetto di *index matching* per i diversi tipi di indice

Index matching

- Diciamo che un indice soddisfa un predicato di selezione se l'indice può essere utilizzato per valutare il predicato

- Assumiamo ad esempio di avere una relazione R(A,B,C,D) e un indice hash multi-colonna sulla chiave composta (A,B)
  - select * from R where A=10 AND B=5 □ **MATCH!**
  - select * from R where A=5 □ **NO MATCH!**

Index matching: indici hash

- Supponiamo di avere la seguente selezione:

$$pred_1 \text{ AND } pred_2 \text{ AND } \ldots$$

- Un indice hash su (A,B,…) soddisfa la condizione di selezione se e solo se **tutti** gli attributi nella chiave di ricerca dell'indice compaiono in un predicato di identità (=)

Index matching: indici hash

- Assumiamo di avere una relazione R(A,B,C,D)
- La tabella qui sotto riporta esempi di *match* e *non match* dell'indice

selection condition	hash index on (A,B,C)	hash index on (B)
A=5 **AND** B=3	no	yes
A>5 **AND** B<4	no	no
B=3	no	yes
A=5 **AND** C>10	no	no
A=5 **AND** B=3 **AND** C=1	yes	yes
A=5 **AND** B=3 **AND** C=1 **AND** D >6	yes	yes

Index matching: indici B+ tree

- Supponiamo di avere la seguente selezione:

$$pred_1 \text{ AND } pred_2 \text{ AND } \ldots$$

- Un indice B+tree su (A,B,…) soddisfa la condizione di selezione se e solo se:
  - gli attributi nei predicati costituiscono un prefisso della chiave di ricerca del B+ tree
  - i predicati possono includere qualsiasi operatore (=, <, >, …)

Index matching: indici B+ tree

- Assumiamo di avere una relazione R(A,B,C,D)
- La tabella qui sotto riporta esempi di *match* e *non match* dell'indice

selection condition	B+ tree on (A,B,C)	B+ tree on (B,C)
A=5 **AND** B=3	yes	yes
A>5 **AND** B<4	yes	yes
B=3	no	yes
A=5 **AND** C>10	yes	no
A=5 **AND** B=3 **AND** C=1	yes	yes
A=5 **AND** B=3 **AND** C=1 **AND** D >6	yes	yes

Index matching

- In alcuni casi, una selezione può matchare più indici
- Ad esempio, assumiamo di avere:
  - Hash index su A
  - B+ tree index su (B,C)
  - Selezione: A=7 AND B=5 AND C=4
- Come possiamo usare gli indici per valutare la condizione di selezione?

Index matching

- Ci sono diverse possibilità:
  - usare l'hash index su A e poi verificare le condizioni B=5 e C=4 solo sulle tuple recuperate
  - usare il B+ tree index su (B,C) e poi verificare la condizione A=7 solo sulle tuple recuperate
  - usare entrambi gli indici, fare l'intersezione dei RID e solo successivamente recuperare le tuple corrispondenti

Index matching: selezione con disgiunzione

- Assumiamo di avere
  - hash index su (A)
  - hash index su (B)
  - selezione: A=7 OR B>5
- In questo caso, solo il primo predicato matcha l'indice
- Unica soluzione: file SCAN

Index matching: selezione con disgiunzione

- Assumiamo di avere
  - hash index su (A)
  - B+ tree index su (B)
  - selezione: A=7 OR B>5

- Possibile soluzione: file SCAN, oppure
- Usare entrambi gli indici separatamente, fare l'unione degli insiemi di RID ottenuti e infine recuperare le tuple corrispondenti ai RID

Index matching: selezione con disgiunzione

- Assumiamo di avere
  - hash index su (A)
  - B+ tree index su (B)
  - selezione: (A=7 **OR** C > 5) **AND** B>5

- Usare il B+ tree index per la seconda condizione (B>5) e poi filtrare le tuple in base alle altre due condizioni (A=7 **OR** C > 5)

Tabella riassuntiva dei costi

Tabella 1: Costi Operazioni su singola relazione

	Scan SELECT * FROM R	EqSearch SELECT * FROM R WHERE R.A = c	Insert INSERT INTO R VALUES (...)	Delete DELETE FROM R WERE R.A = c
Heap file	P_R	P_R	$1+1$	$P_R + 2 \cdot \lvert R_{A=c} \rvert$
Sorted file	P_R	$\lceil \log_2 P_R \rceil + \left\lceil \frac{\lvert R_{A=c} \rvert}{\lfloor \frac{P}{t_R} \rfloor} \right\rceil$	$\lceil \log_2 P_R \rceil + 1$	$\lceil \log_2 P_R \rceil + 2 \cdot \left\lceil \frac{\lvert R_{A=c} \rvert}{\lfloor \frac{P}{t_R} \rfloor} \right\rceil$
Unclustered B$^+$-tree	P_R	$L_b + \lvert R_{A=c} \rvert$	$L_b + 1 + 1$	$L_b + 2 \cdot \lvert R_{A=c} \rvert$
Clustered B$^+$-tree	P_R	$L_b + \left\lceil \frac{\lvert R_{A=c} \rvert}{\lfloor \frac{P}{t_R} \rfloor} \right\rceil$	$L_b + 1 + 1$	$L_b + 2 \cdot \left\lceil \frac{\lvert R_{A=c} \rvert}{\lfloor \frac{P}{t_R} \rfloor} \right\rceil$
Hash index	P_R	$L_h + \lvert R_{A=c} \rvert$	$L_h + 1 + 1$	$L_h + 2 \cdot \lvert R_{A=c} \rvert$

Tabella 2: Costi algoritmi di join. Relazioni R(A,B) e S(C,D).

	Costo di $R \underset{R.B=S.C}{\bowtie} S$
Nested loop join	$P_R + P_R \cdot P_S$
Sort-merge join	$\text{Costo}_{\text{OrdR}} + \text{Costo}_{\text{OrdS}} + P_R + P_S$
Hash join	$(P_R + 2 \cdot \lvert R \rvert) + (P_S + 2 \cdot \lvert S \rvert) + (P_R + P_S)$
Index Nested Loop (indice su S.C)	$P_R + \lvert R \rvert \cdot \text{Costo}_{\text{EqSearchS}}$