Innanzitutto, cerchiamo di dare una spiegazione di come funzionano gli indici effettivamente e qual è il loro utilizzo. Consideriamo una tabella in cui dobbiamo scansionare una grande serie di righe e di colonne. Con grandi moli di dati, ciò potrebbe essere particolarmente dispendioso.
Abbiamo quindi bisogno di parallelizzare o strutturare questa ricerca per poterla accelerare, per esempio anche frammentando tutto in base alle dimensioni delle chiavi.
Quello che andiamo fare con gli indici è avere una piccola struttura dati che ci permetta di saltare direttamente dove ci interessa.

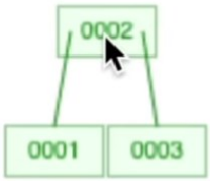
Possiamo usare strutture dati ausiliarie come gli alberi, per esempio gli alberi binari, facendo in modo che noi abbiamo una ricerca per un valore e:

- Se abbiamo un valore più piccolo di quello attuale, cerchiamo a sinistra
- Se abbiamo un valore più grande di quello attuale, cerchiamo a destra

Normalmente, da Algoritmi, sappiamo che questo ha un costo logaritmico, se infatti non sappiamo esattamente dove cercare e scansioniamo tutto l'albero.
Il problema è questo: *non sono bilanciati*.
Se capitasse di dover cercare un valore da qualche parte, per pure sfortuna, potremmo avere dei casi in cui cerchiamo un valore molto grande e la nostra ricerca potrebbe, col tempo, diventare lineare, per esempio continuando a cercare in un solo ramo dell'albero.

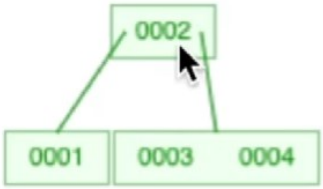
Partiamo quindi dal B-Tree, che è un albero bilanciato, cioè si bilancia mentre i dati vengono inseriti; tutto ciò ha un costo, continuando a riordinare l'albero elemento per elemento. Si consideri che gli alberi bilanciati presentano, per ogni nodo, una coppia di elementi
Partiamo ad inserire elementi (letteralmente, i numeri indicati tra le parentesi tonde):

(1) → Sarà la radice
(2) → Sarà inserito nello stesso nodo della radice
(3) → L'albero assomiglierà a:



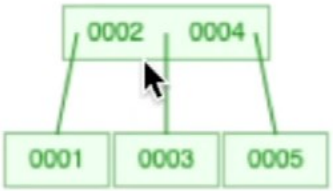
L'elemento più grande diventa la foglia destra (3), quello più piccolo lo foglia sinistra (1), e (2) resta in mezzo.

(4) → L'albero assomiglierà a:



(2) resta radice, (4) si inserisce a destra, (1) rimane foglia a sinistra, ma ora (3) avrà il puntatore a (4).

(5) → L'albero assomiglierà a:



Si nota che l'albero continua a sistemarsi, mettendo a coppie gli elementi più piccoli a sinistra, tenendo gli elementi più grandi a coppie a destra e via così.

Che importanza ha tutto questo? Nei database, questi sono pagine. A me non interessa cercare un valore, ma un contenuto associato al valore (puntando o a un valore di tupla nel caso di MySQL e in Postgre si usano campi secondari). Nel caso dei B-Tree abbiamo che ogni chiave contiene un dato e, dato che l'albero si bilancia, i dati accessi più di frequente tenderanno ad accumularsi vicino alla radice.

Il fatto è che, nei B-Tree, nei nodi interni e foglia sono inserite le chiavi assieme ad altri dati; se dovessimo cercare un valore non spesso cercato, dovremmo cercare nelle foglie intermedie, risultando parecchio costoso. Se invece dovessimo cercare un gruppo/insieme di valori, dovremmo percorrere più volte le varie foglie, risultando decisamente sconveniente.
L'albero tende a bilanciarsi, come si vede, sulla base dei valori inseriti ma:

- Con gruppi di valori → Maggior costo nella ricerca dell'albero
- Con valori cercati non troppo spesso

Quindi, ad ogni operazione, tocca cercare molti più nodi e valori.

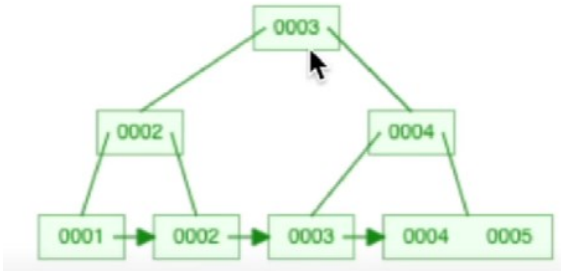
Nell'esempio di prima, infatti, consideriamo di dover cercare 1/3/5 allo stesso tempo:

Dovremmo:
- Saltare a sinistra per trovare (1)
- Saltare al centro per cercare (3)
- Saltare a destra per cercare (5)

Totale: 3 I/O per arrivare a 5. Dovessimo cercare 100, ad esempio, dobbiamo guardarli tutti, pur essendo tutti vicini.

Quindi, mettendo i valori di fianco alle chiavi, sprechiamo spazio.

Se per esempio, mettessimo *i valori assieme alle chiavi solo nelle foglie*, avremmo i B+Tree (un esempio a fianco).
Una prima convenienza: cercassimo un valore vicino ad un'altra chiave, avremmo subito tutto il range di valori con una sola ricerca.



Esempio pratico: cercare tutti i valori da 1 a 5 (con le imm. di esempio):
- Nel B-Tree, dovremmo guardali tutti, da 1 a 5
- Nel B+Tree, si trova 1 e si ha trovato tutto il resto subito, dato che abbiamo subito tutti i puntatori, di cui non ci interessa il valore del nodo, ma invece con un valore abbiamo subito tutto il range.

Si ha anche che se dovessimo cercare su un solo valore (ma non su un range), si potrebbe semplicemente usare un B-tree, che comunque rimane limitato nell'utilizzo.

Vantaggi dei B+Tree:
- Poiché i B+Tree non hanno dati associati ai nodi interni, è possibile inserire più chiavi in una pagina di memoria. Pertanto, saranno necessari meno missaggi alla cache per accedere ai dati che si trovano su un nodo foglia.
- I nodi foglia dei B+Tree sono collegati, quindi una scansione completa di tutti gli oggetti di un albero richiede un solo passaggio lineare attraverso tutti i nodi foglia. Un B-Tree, invece, richiede l'attraversamento di ogni livello dell'albero. Questo attraversamento completo dell'albero comporterà probabilmente un maggior numero di miss della cache rispetto alla traversata lineare delle foglie di B+.

Vantaggi dei B-Tree:
- Poiché i B-Tree contengono dati per ogni chiave, i nodi a cui si accede di frequente possono trovarsi più vicini alla radice e quindi possono essere consultati più rapidamente.

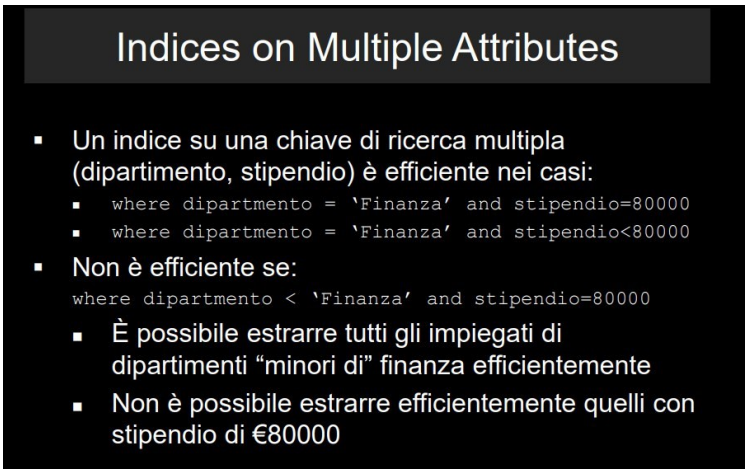
Nei B-Tree, per questo motivo, spesso si attraversa tutto l'albero, che quindi deve rimanere tutto in memoria e comporta un costo molto grande, rispetto ai B+ Tree, dove semplicemente si va alle foglie, non dovendo guardare sempre tutti i nodi per poterlo bilanciare come capita nei B-Tree e quindi tenerli tutti in memoria (in RAM, caso MongoDB che possiede indici B-Tree, consumando molta memoria).

Il vantaggio principale dei B+Tree rispetto ai B-Tree è che consentono di inserire più puntatori ad altri nodi rimuovendo i puntatori ai dati, aumentando così il numero di input e diminuendo potenzialmente la profondità dell'albero.
Lo svantaggio è che non ci sono uscite anticipate quando si potrebbe trovare una corrispondenza in un nodo interno. Ma poiché entrambe le strutture di dati hanno un numero di input enorme, la stragrande maggioranza delle corrispondenze si troverà comunque sui nodi foglia, rendendo in media più efficiente.

Altro esempio (avendo come informazioni delle date e ragionando sugli anni e mesi):
01/18 02/18 07/18 01/19 05/19

- Se tu hai un indice così è gli dici "dammi quello che hanno anno>19" lui deve percorrerlo tutto in quanto gli anni son sparsi dentro a ogni mese
- Se invece dici "mese =02 e anno >17" salta subito al mese 02, e poi trova gli anni sopra al 17

Altro esempio ancora:



- Qua sul primo salta subito a "finanza" e cerca i <800
- Nel secondo invece deve passare tutti i dipartimenti e trovare quelli < di finanza

L'ordine dei capi di un indice conta. È come se fosse un ORDER BY:

- Se ti fai "order by year month" quello che fa è prima ordina per anno, e poi ordina (dentro ogni anno) per mese
- Se fai "order by month year" invece prima ordina per mese e poi per anno, quindi prima ordina tutto per mese e poi dentro ogni mese ordina gli anni

Prendiamo, infine, il caso delle tabelle hash.

In una hashtable è possibile accedere agli elementi solo in base alla loro *chiave primaria* in tempo *costante*. Questo è più veloce rispetto a un algoritmo ad albero (O(1) invece di log(n)), ma non è possibile selezionare intervalli (tutto ciò che è compreso tra x e y).
Gli algoritmi ad albero supportano questa operazione in Log(n), *mentre gli indici hash possono portare a una scansione completa della tabella O(n)*. Anche l'overhead costante degli indici hash è di solito maggiore Inoltre, gli algoritmi ad albero sono solitamente più facili da mantenere, crescere con i dati, scalare, ecc.

Gli indici hash funzionano con dimensioni hash predefinite, quindi si finisce per avere alcune "scatole" in cui sono memorizzati gli oggetti. Questi oggetti vengono ripetuti in loop per trovare quello giusto all'interno di questa partizione.
Quindi, se le dimensioni sono piccole, si ha un notevole overhead per gli elementi piccoli, mentre le dimensioni grandi comportano un'ulteriore scansione.
Gli algoritmi delle tabelle hash di oggi sono generalmente scalabili, ma la scalabilità può essere inefficiente.

La differenza tra l'uso di un b-tree e di una tabella hash è che:

- il primo consente di utilizzare i confronti tra colonne nelle espressioni che utilizzano gli operatori =, >, >=, <, <= o BETWEEN
- il secondo è utilizzato solo per i confronti di uguaglianza che utilizzano gli operatori = o <=>.

Buona tabella riassuntiva di confronto:

B+ Tree	mediamente bene per = consigliati per intervalli < o > bene per cancellazione o inserimento
B- Tree	mediamente bene per = male per cancellazione o inserimento mediocri in generale
Hash	migliori per uguaglianze male ORDER BY

uguaglianza	Hash
intervallo	B+ Tree
uguaglianza e intervallo	B+ Tree
cancellazione/inserimento	B+ Tree
ORDER BY	B+ Tree

Detto questo, commentiamo una serie di esercizi e la motivazione della risposta corretta:

Data la relazione e R(A, B, C) la query SELECT * FROM R ORDER BY C, quale dei seguenti indici velocizza l'esecuzione della query?

(1) Indice Hash su C

(2) Indice B+Tree su C

(3) Indice Hash su A, B, C,

(4) Indice B+Tree su A, B, C

La risposta corretta è la (2), dato che dovendo scegliere tra B+ Tree ed Hash, sappiamo che andremo a scegliere la seconda solo se dovessimo cercare un singolo valore.
Qui occorre selezionare ogni singolo campo ed ordinarlo per C; dato che dobbiamo prendere un insieme di valori e confrontarne *ognuno* per C, allora, useremo un indice B+ Tree. Questo viene fatto su C dato che confronta ogni singolo valore e, trovato C, svolge l'operazione in poco tempo.

Data la query `SELECT * FROM R WHERE C='Valore'` sulla relazione R(A, B, C). Quale dei seguenti indici in genere assicura le migliori performance in termini di velocità dell'esecuzione della query?

(1) Indice Hash su C;

(2) Indice B-Tree su C;[3]

(3) Indice Hash su A, B, C;

(4) Indice B-Tree su A, B, C.

La risposta corretta è evidentemente la (1), dato che qui non devo confrontare valori ma solo e semplicemente cercare *Valore* per C. Questo porta a scegliere per certo la Hash, in grado di cercare in tempo costante O(1) il valore di interesse.

Data la query `SELECT * FROM S WHERE X=4 AND Z>8` sulla relazione `S(X, Y, Z, W)`. Quale dei seguenti indici in genere assicura le migliori performance in termini di velocità dell'esecuzione della query?

1. Indice Hash su (Z,X)
2. Indice B+Tree su X
3. Indice Hash sulla coppia (X,Z)
4. Indice B+Tree sulla coppia (X,Z)

La risposta corretta è la (4), dato che dobbiamo confrontare a coppie due valori. Si consideri, inoltre, che l'ordine deve essere esattamente X e Z, dato che:

- su X si ha un'uguaglianza e conviene, da un punto di vista di albero, cercare il nodo con X subito
- trovato X, si indicizzano tutti gli altri campi sulla base di Z.

Operano confronti, si sceglie B+ Tree e proprio sulla coppia (X, Z); per il motivo spiegato, non si potrebbe scegliere (Z, X).

Data la query `SELECT * FROM S WHERE Z=4 ORDER BY X` sulla relazione `S(X, Y, Z, W)`. Quale dei seguenti indici in genere assicura le migliori performance in termini di velocità dell'esecuzione della query?

1. Indice B+Tree sulla coppia (X,Z)
2. Indice B+Tree su X
3. Indice Hash sulla coppia (X,Z)
4. Indice B+Tree sulla coppia (Z,X) [3]

La scelta ricade sulla risposta (4), dato che anche qui confrontiamo una coppia di valori. Per il ragionamento appena fatto e parte di un esempio sopra, conviene trovare la foglia che presenta il campo uguale a Z e,

trovato quello, ordinare ogni valore per X che sarà il secondo valore (prima pongo la condizione, poi ordino sul campo, ragionando così).

Data la query `SELECT * FROM S WHERE Z>4 AND X=5` sulla relazione `S(X, Y, Z, W)`. Quale dei seguenti indici in genere assicura le migliori performance in termini di velocità dell'esecuzione della query?

1. Indice B+Tree sulla coppia (X,Z)
2. Indice Hash sulla coppia (Z,X)
3. Indice Hash sulla coppia (X,Z)
4. Indice B+Tree sulla coppia (Z,X)

Questa sembra essere uguale a 2 risposte fa, ma si nota che l'uguaglianza viene effettuata su X. Dato che dobbiamo cercare e confrontare una serie di valore, indubbiamente, si sceglie il B+ Tree. Come detto, conviene ordinare ogni foglia per X, avendo un'uguaglianza e cercare a coppie per Z.
Per questo motivo, la risposta corretta sarà proprio la (1).

Si consideri le relazioni R(A, B, C, D) e la seguente query

SELECT MIN(A) FROM R WHERE B=10

Quale dei seguenti indici garantisce l'efficienza massima?

1. Indice Hash sulla coppia (B,A)
2. Indice Hash sulla coppia (A,B)
3. Indice B-TREE sulla coppia (A,B)
4. Indice B-TREE sulla coppia (B,A)

Qui usiamo una funzione di aggregazioni, confrontando ogni valore per B. Se non avessimo la funzione di aggregazione, potremmo usare una hash su B.
Tuttavia, conviene usare un B-Tree. Dato che ogni singolo valore va ordinato per minimo sulla base di B, allora conviene ordinare per B e selezionare successivamente per A.
Per tutte queste motivazioni, la risposta corretta è la (4).

Si consideri le relazioni R(A, B, C, D) e la seguente query

SELECT MIN(B) FROM R WHERE A=10

Quale dei seguenti indici garantisce l'efficienza massima?

1. Indice Hash sulla coppia (B,A)
2. Indice Hash sulla coppia (A,B)
3. Indice B-TREE sulla coppia (A,B)
4. Indice B-TREE sulla coppia (B,A)

Stesso ragionamento visto sopra, solo che si inverte B con A.

Data la query `SELECT * FROM S WHERE A=4 ORDER BY B` sulla relazione `S(A,B,C)`. Quale dei seguenti indici in genere assicura le migliori performance in termini di velocità dell'esecuzione della query?

1. Indice B+Tree sulla coppia (B,A)
2. Indice B+Tree su B
3. Indice B+Tree su A
4. Indice B+Tree sulla coppia (A,B)

Confronto su una coppia di valori e per certo si sceglie B+Tree (non avremmo comunque altre opzioni di scelta qui volendo). Dobbiamo scegliere tra A, B o entrambi. Si nota che sussiste una ricerca per A ed un successivo ordinamento per B. Si va, quindi, a selezionare ogni campo di B sulla base del valore trovato per A, come detto in tutti i casi sopra, per velocità computazionale e costo.
Quindi, la risposta corretta è la (4).