Index-uri cu structură dinamică – arbori de căutare de tip B+. Formatul

nodurilor. Algoritmi de căutare şi de inserare. Exemple.

(text preliminar)

Anterior conceperii structurilor dinamice de indexare în domeniul bazelor de date, una din metodele cele mai răspândite pentru realizarea accesului la informație a fost cea a **accesului secvențial indexat** (ASI). Această metodă, dezvoltată inițial în mediul IBM cu intenția extragerii rapide a informațiilor memorate de calculatoarele tip „mainframe", echipate la început cu memorii externe pe benzi magnetice, continuă ă fie folosită încă pe scară largă în sistemele de baze de date, fie ele relaționale sau nu.

Metoda ASI prevede **memorarea secvențială a înregistrărilor de fişier**, ceea ce era normal în cazul folosirii de benzi magnetice. Extragerea înregistrărilor individuale se realizează cu ajutorul unui sistem de indicatori de adresă („pointeri") orientați către înregistrări. Indicatorii de adresă sunt memorați în tabele de dispersie („tabele hash"), tabele ce poartă numele de index-uri. Această manieră de acces permite extragerea înregistrărilor fără a fi necesar un proces de căutare în întregul volum de informație.

Pe de altă parte, vom remarca faptul că în bazele de date de tip „navigațional", pointerii către alte informații erau memorați chiar în interiorul înregistrărilor din care se inițiază procesul de căutare. Avantajul utilizării metodei ASI constă în lungimea redusă a index-urilor şi, prin urmare , în timpul scurt de căutare la acest nivel. În acest mod, baza de date va efectua operațiuni de acces numai la înregistrările dorite.

În contextul folosirii metodei ASI, realizarea bazelor de date relaționale impune menținerea valabilității legăturilor între relații (tabele), deziderat care se realizează prin mijloace logice corespunzătoare. În practică, pentru o căutare rapidă, se indexează un element (câmp) din înregistrare (tuplu), ce poartă numele de cheie externă (străină). În această abordare, viteza de acces va fi mai mică decât atunci când pointerii către datele căutate sunt depuşi direct în înregistrări. Pe de altă parte, modificările în aranjarea fizică a datelor nu fac necesară actualizarea pointerilor.

Întrucât **metoda ASI** constă în **accesul secvențial direct la fişierele bazei de date,** implementarea sa, ca şi înțelegerea ei conceptuală, nu ridică probleme – în majoritatea cazurilor, fapt ce conduce şi la un cost redus al folosirii metodei. Totuşi, în unele situații, ca, de exemplu, în configurațiile client-server, fiecare sistem client trebuie să gestioneze propria sa legătură cu fiecare fişie la care realizează un acces, fapt care face posibile operațiuni conflictuale de inserare de tupluri în aceste fişiere şi, în consecință, stări de incoerență a bazei de date. Problema menționată este soluționată de sistemele curente de gestiune a bazelor de date, care conțin un modul de reglementare client-server.

Ulterior, pentru înlocuirea metodei ASI, a fost propusă tehnica metodei de acces cu memorie virtuală (IBM), care este metoda de acces fizic la informație folosită în sistemul DB2.

Structuri dinamice de index. Arbori B+.

ASI - inconvenient- lanţuri lungi de "overflow" apar când fişierul creşte, deci -> performanţe mai slabe. Consecinţă-> au apărut structuri dinamice, mai flexibile, care se adaptează uşor operaţiilor de inserare şi eliminare. Una din aceste structuri: arborii de căutare B+. În esenţă, arborele B+ este un arbore balansat , în care nodurile interne direcţionează procesul de căutare, iar nodurile frunză (terminale) conţin intrările de date.

În modelul ASI, alocarea (adăugarea) de pagini frunză se făcea static, secvenţial. În cazul arborilor B+, structura arborilor se dezvoltă (creşte) şi se reduce în mod dinamic, deci nu este posibil ca paginile frunză să fie alocate secvenţial-static. Pentru a extrage paginile frunză în mod eficient (deci rapid! De fapt, este vorba de extragerea informaţiilor din pagină, ceea ce presupune extragerea paginii din memoria secundară -> cea centrală), acestea trebuie legate (între ele) prin pointeri de pagină. O cale de a lega paginile este organizarea lor în o listă dublu înlănţuită. În acest mod, vom putea traversa secvenţa paginilor în orice direcţie . O structură de acest gen – mai jos - în fig. 1.

ASI este o structură "statică". In cazul structurii dimnamice:

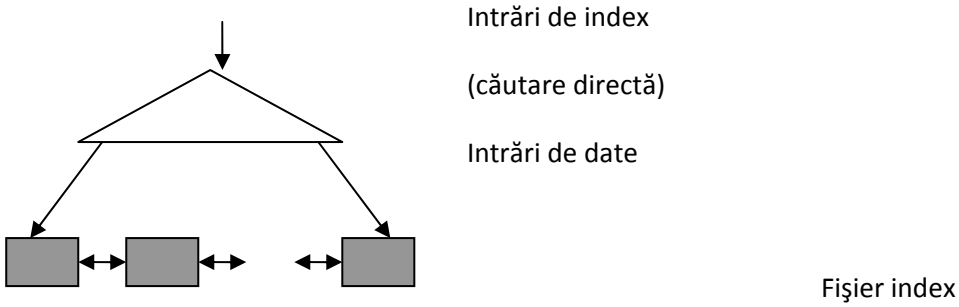


Fig. 1 Structura unui arbore B+

Printre caracteristicile fundamentale ale arborilor B+:

- Operaţiile (inserare, eliminare) menţin echilibrul arborelui.
- Grad de ocupare de minimum 50% garantat pentru fiecare nod. Excepţie – nodul rădăcină, dacă se foloseşte un algoritm de eliminare specific. Remarcăm, totuşi, că eliminarea se implementează adesea prin simpla localizare a intrării de date şi a eliminării acesteia, fără ajustarea arborelui, aşa cum ar cere garantarea unui grad de ocupare de 50%. Cauza – arborii cresc de obicei, nu se diminuează.
- Căutarea unei înregistrări – reclamă numai o traversare a arborelui de la rădăcină până la frunza corespunzătoare. Înălţimea arborelui este lungimea unei căi de la rădăcină până la o frunză (până la orice frunză, arborele fiind echilibrat. Arborii B+ au coeficient de ramificare la ieşire ("fan-out") mare, de aceea înălţimea lor este rareori > ca 3 sau 4.

Pentru arborii B+ notăm:

- d = ordinul arborelui B+, care este o măsură a capacității unui nod al arborelui
- m = numărul de intrări în fiecare nod – pentru arborii B+ pe care îi studiem aici

În general, avem: d<= m<=2d.

Pentru nodul rădăcină: 1<=m<=2d.

Considerații calitative:

În general, este mai avantajos să folosim un index cu arbore B+ cu înregistrări de date memorate ca intrări de date, în loc de a folosi un fişier sortat. Pentru "overhead"-ul de spațiu necesitat de memorarea intrărilor indexului, se obțin avantajele fişierelor sortate, la care se adaugă algoritmi eficienți de inserare şi eliminare. De obicei, arborii B+ asigură cca 67% grad de ocupare a spațiului de memorie.

Arborii B+ sunt preferați, de obicei, indexării ASI, deoarece asigură inserarea fără înlănțuiri de "overflow".

Remarcăm, totuşi, că dacă dimensiunile şi distribuirea volumului de date rămân îndeajuns de "statice", înlănțuirile de "overflow" pot să nu constituie o dificultate. în astfel de cazuri, AIS este recomandabilă.

Regulă generală: arborii B+ au performanțe mai bune ca AIS.

<u>Formatul unui nod</u>

Este acelaşi cu formatul unui nod (pagină de index!) AIS– vezi fig. 2

fig. 2

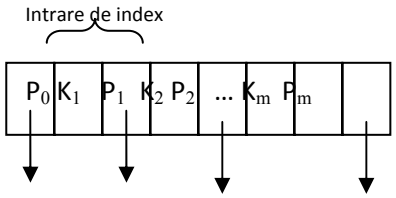


Fig. 2

Formatul unei pagini de index.

Aici: nodurile interne cu m intrări de index conțin m+1 pointeri către succesori (copii). Pointerul P_i – indică spre un subarbore în care toate valorile de cheie K sunt a.î.: $K_i<=K<K_{i+1}$.

Cazuri speciale:

P_0 – indică spre un arbore în care toate $K<K_1$

P_m – indică spre un arbore în care toate $K>=K_m$

Pentru nodurile frunză (terminale), intrările sunt notate K* - uzual. În arborii B+, nodurile frunză (şi numai ele!) conțin "intrări de date" ("data entries"). În cazul obişnuit al folosirii alternativelor (2) şi (3), intrările frunză sunt perechi <K, I(K)>, exact ca şi intrările interne (ne-terminale). Mai observăm că indiferent de alternativa aleasă pentru intrările frunză, paginile frunză sunt înlănțuite în o listă dublu înlănțuită. Aşadar, frunzele formează o secvență, care poate fi folosită pentru a răspunde eficient la interogările care primesc intervale de valori.

Este interesant de a analiza cum se poate ajunge la o organizare a nodului de arbore ca cea de mai sus, folosind alte formate de înregistrări (cu lungime fixă, cu lungime variabilă). De fapt, fiecare pereche cheie-pointer poate fi văzută ca o înregistrare. Dacă fişierul care se indexează este de lungime fixă a înregistrărilor, intrările din index vor avea lungime fixă. În cazul contrar, ar fi de lungime variabilă. În ambele cazuri, arborele B+ poate fi văzut el însuşi ca un fişier de înregistrări. Dacă paginile frunză (terminale) nu conțin înregistrările autentice de date, atunci arborele B+ este efectiv un fişier de înregistrări, adică distinct de fişierul care conține datele. Dacă însă paginile frunză (terminale) conțin înregistrări de date, atunci un fişier conține atât arborele B+ cât şi datele.

Procesul de căutare

Algoritmul de căutare găseşte nodul frunză căruia îi aparține o intrare de date anumită. În cele ce urmează (ca şi pentru algoritmul de inserare şi eliminare pentru arborii B+) se presupune că nu există duplicate. Cu alte cuvinte, nu pot exista două intrări de date cu aceeaşi valoare a cheii. În practică, pot apărea duplicate.

Notații folosite:

Ptr – variabilă pointer;

* ptr – valoarea către care indică pointerul ptr

& (valoare) – adresa pentru valoare

Vom remarca faptul că, pentru a găsi i în procesul de căutare în arbore, trebuie să efectuăm o căutare în interiorul nodului, ceea ce se poate realiza fie prin o căutare liniară, fie prin o căutare binară (în funcție de numărul de intrări în nod).

Algoritmul de căutare – prezentat în pseudocodul care urmează:

func găseşte (valoarea K a cheii de căutare) returns pointer-de-nod

//Fiind dată o valoare de cheie de căutare, găseşte nodul sau frunza

return căutare-în-arbore (rădăcina, K);

end func

func căutare-în-nod (pointer-de nod, valoare K a cheii de căutare) returns pointer-de-nod

//Caută în arbore pentru intrare

*if *pointer-de-nod este o frunză, return pointer-de-nod;*

else,

if K<K_1 then return căutare-în-arbore(P_0, K);

else,

if K>=K_m then return căutare-în-arbore (P_m, K);//m= #intrări

else,

find i astfel încât K_i<=K<K_{i+1};

return căutare-în-arbore (P_i,K)

end func

Fig. 3 Algoritm pentru căutare – pseudocod

Pentru a ilustra funcționarea (parcurgerea) algoritmului de căutare – fie exemplul arborelui B+ din Fig. 4. Remarcăm că ordinul arborelui este d=2, adică fiecare nod conține între 2 şi 4 intrări. Fiecare intrare internş (ne-frunzş !) este – conține – o pereche <valoare de cheie, pointer-de-nod>. La nivelul frunzelor (nodurilor terminale), intrările sunt înregistrări de date, notate cu R*.

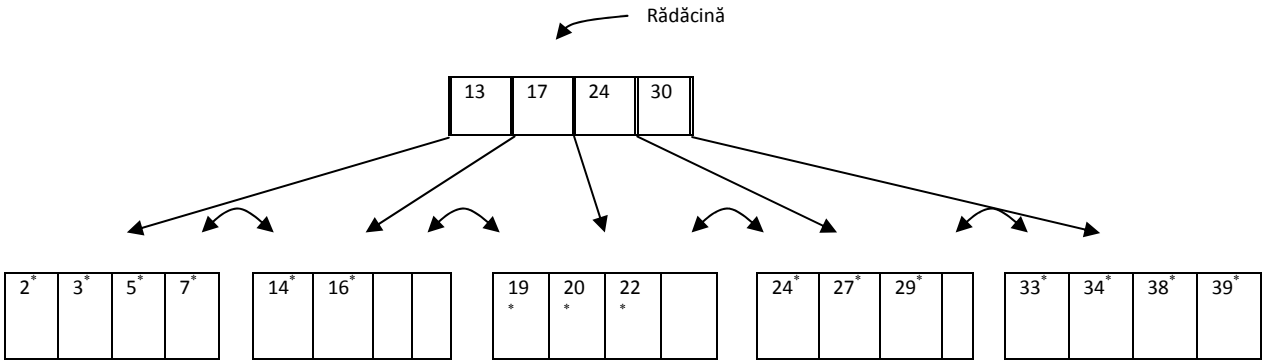


Fig. 4 Arbore B+ de ordinul d=2

Presupunem că dorim să efectuăm o căutare pentru intrarea 5*. În acest scop, vom urmări pointerul succesorului (copilului) din extrema stângă, deoarece 5<13. Dacă vom căuta intrările 14* sau 15*, urmărim al doilea pointer, deoarece 13<=14<17, iar 13<=15<17. Deoarece nu găsim în paginile frunză 15*, putem trage concluzia că această valoare nu este prezentă în arbore! Pentru a găsi 24*, vom folosi pointerul celui de al patrulea succesor (copil), deoarece 24<=24<30.

Procesul de inserare

Algoritmul de inserare la o intrare (din index), găseşte nodul frunză căruia îi aparține şi inserează "acolo" înregistrarea.

Ideea algoritmului de înserare stă în aceea că inserăm recursiv o intrare apelând algoritmul de inserare la nivelul nodului succesor (copie) corespunzător. De obicei, această prodcedură ne conduce "jos", la frunza căreia îi aparține intrarea, amplasând intrarea acolo şi revenind apoi la nodul rădăcină.

Remarcăm că uneori un nod este complet ("plin") şi, ca urmare, trebuie secționat. Atunci când este secționat nodul, în părintele său trebuie inserată o intrare care va indica spre nodul creat prin secționare. Către această intrare, la rândul său, va indica o "variabila pointer" numită "copii-noi", sau succesori noi.. Prin secționarea vechiului nod şi crearea unui nou nod rădăcină, înălțimea arborelui creşte cu o unitate.

Pseudocodul algoritmului - În Fig. 5. Pentru a ilustra procesul de inserare, vom folosi arborele prezentat anterior – Fig. 4, pentru procesul de căutare.

Admitem că dorim să inserăm intrarea 8*. Aceasta aparține, evident, frunzei (nodului frunză) din extrema stângă - care este deja plin. Potrivit algoritmului, procedura de inserare va efectua o secționare a paginii – frunză. Paginile rezultate prin secționare – sunt prezentate în Fig. 6. Întregul arbore trebuie acum

“ajustat”, pentru a lua în considerație noua pagină frunză care a apărut. În acest scop, inserăm o intrare care constă din perechea <5, pointer către pagina nouă>, în nodul părinte. Aici, observăm în ce nod este copiată cheia 5, care descrie diferența între pagina frunză secționată şi sora (sau “fratele”) său creată. Nu putem ca “pur şi simplu”, să “împingem” (deplasăm) în sus (“push up”) valoarea 5, deoarece fiecare intrare de date trebuie să apară în o pagină frunză.

Deoarece nodul părinte este, de asemenea, plin, apare o altă secționare. În general, trebuie să secționăm un nod ne-frunză atunci când este plin şi conține 2d chei şi 2d+1 pointeri, respectiv trebuie să adăugăm o altă intrare de index care să corespundă unei secționări de nod copil. Avem, în acest moment, 2d+1 chei şi 2d+2 pointeri, cu două noduri ne-frunză pline (în mod minimal!!), fiecare din acestea conținând d chei şi d+1 pointeri. De asemenea, mai avem o cheie în plus, pe care o facem cheie “de mijloc” (mediană). Această cheie şi un pointer către al doilea nod ne-cheie constituie o intrare de index care trebuie inserată în părintele nodului ne-frunză secționat. În acest fel, cheia “de mijloc” este “împinsă” (deplasată) spre partea de sus a arborelui (“pushed-up”), spre deosebire de cazul secționării unei pagini frunză.

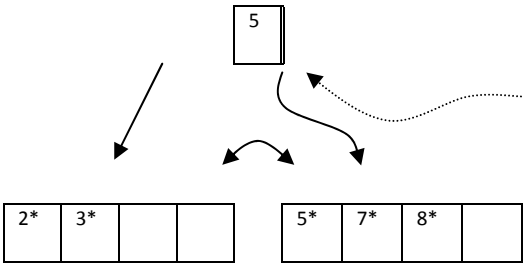
Paginile secționate din exemplul tratat se văd în Fig. 7 . Intrarea de index care indică spre noul nod ne-frunză este perechea <17, pointer spre noua pagină la nivel de index>; trebuie remarcat că valoarea 17 a cheii este “împinsă în sus” pe arbore, spre deosebire de valoarea 5 a cheii de secționare în secțiunea frunzei, care a fost “copiată”în partea superioară.

```
Proc inserează (pointer-de-nod, intrare, copii-noi)
//Inserează o intrare într-un sub-arbore cu rădăcina '* pointer-de-nod';
//gradul este d; 'copii-noi' este inițial nulă, şi tot nulă la return dacă
//nu este secționat un nod copil
if * pointer-de-nod este un nod ne-frunză, să spunem (fie) N,
    find i astfel încât Ki<=valoarea cheii intrării<Ki+1; //alege un subarbore
    insert (Pi, intrare, copii-noi);                    //recursiv, inserează o înregistrare
    if copii-noi is null, return;          //caz uzual; nu s-a secționat nod copil
    else,                          //am secționat nodul copil, trebuie inserat * copii-noi în N
        if N are spațiu,
            pune * copii-noi în el, pune copii-noi la zero, return;
        else                       //notează diferența wrt secționarea paginii frunză!
            secționează N://2d+1 valori de cheie şi 2d+2 pointeri-de-nod
            primele d valori de cheie şi d+1 pointeri-de-nod rămân,
```

```
        ultimele d chei şi d+1 pointeri se deplasează la un nod nou, N2;
        //* copii-noi pus pentru orientarea căutărilor între N şi N2
        copii-noi = &(<cea mai mică valoare de cheie la N2, pointerul către N2>);
        if N este rădăcina,          //rădăcina a fost secționată adineauri
                crează un nod nou cu <pointerul către N, *copii-noi>;
                forțeză pointerul de nod al rădăcinii arborelui să indice spre noul nod;
        return;
if * pointer-de-nod este un nod frunză, fie L,
        if L are spațiu                    //caz uzual
        pune o intrare în el, pune copii-noi la zero, return;
        else,
                secționează L: primele d intrări rămân, restul se
                        deplasează spre nodul nou L2;
                copii-noi = & (<cea mai mică valoare de cheie la L2,
                                                pointerul către L2>);
                pune pointerii frați în L şi L2;
                return
endproc.
```

Fig. 5, Algoritm pentru inserare în arbore B+ de ordin d

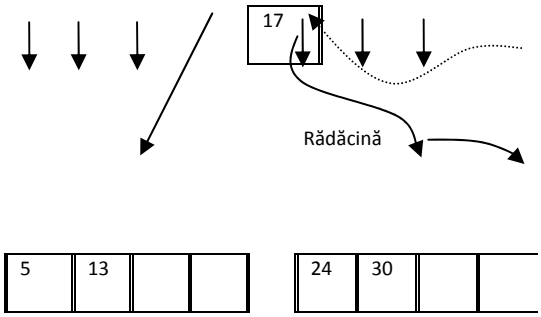
Diferența în tratarea secționărilor la nivel de frunză şi la nivel de index, provine din cerința ca la arborii B+, toate intrările de date R* să fie amplasate în frunze. Această cerință ne împiedică să "împingem în sus" valoarea 5 şi conduce la o uşoară redundanță, având unele valori de cheie care apar atât la nivel de frunză, cât şi la nivel de index. Totuşi, la interogările care solicită valori pe interval ("range queries") , se poate răspunde eficient prin simpla extragere a secvenței de pagini frunză; redundanța este un preț mic pentru eficiența realizată. În procesul de tratare a nivelurilor de index, avem o flexibilitate mai mare, şi "împingem în sus valoarea 17, pentru a evita prezența a două copii ale valorii 17 la nivel de index.



Intrare care se inserează într-un nod părinte (De notat că 5 este copiată "în sus" şi continuă să apară in frunză).

Fig. 6 Secționarea paginilor frunză în procesul

inserării intrării 8*



Intrare care se inserează într-un nod părinte (De notat că 17 este "împinsă în sus" şi că apre odată în index. De văzut aceasta în contrast cu secționarea unei frunze).

Fig. 7 Secționarea paginilor de index în procesul

de inserare a intrării 8.

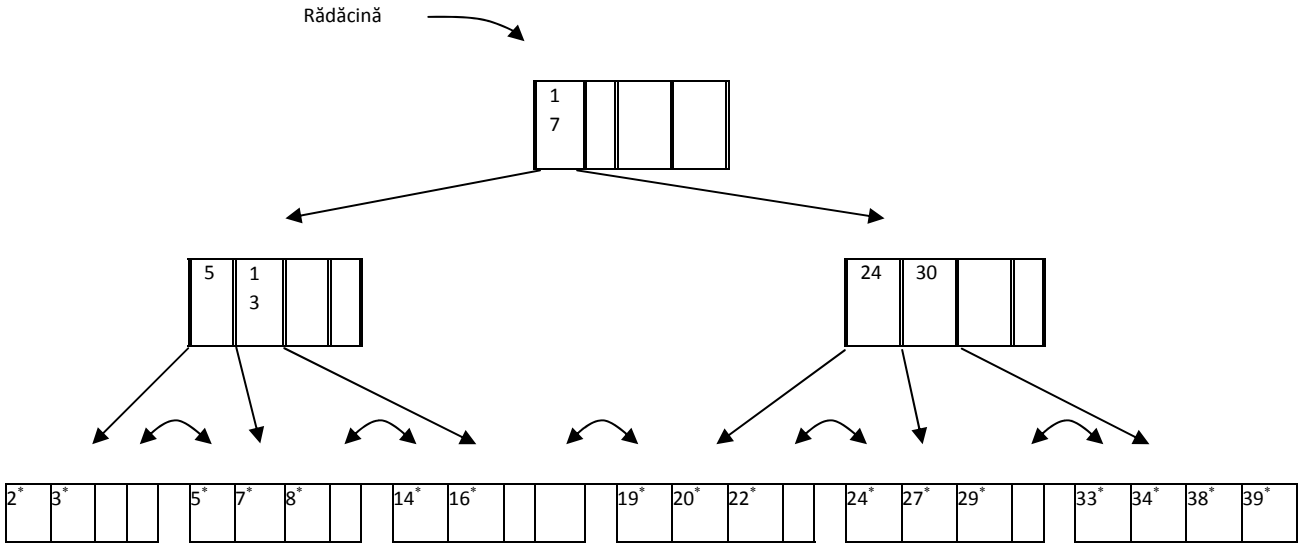


Fig. 8 Arborele B+ după inserarea intrării 8*

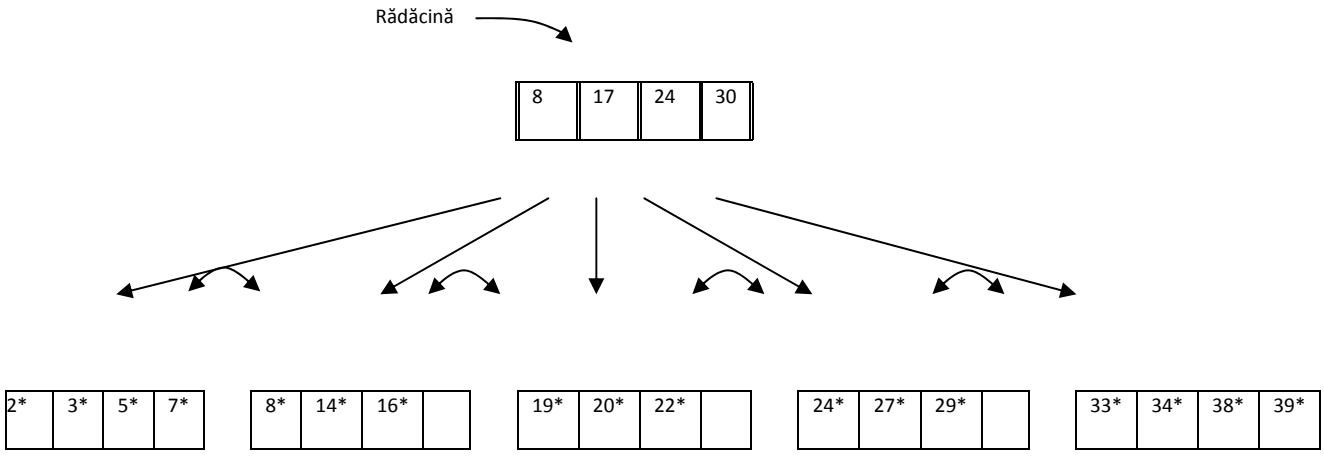


Fig. 9 Arborele B+ după inserarea intrării 8*

folosind redistribuirea

În continuare, deoarece nodul de secționare a fost vechea rădăcină, trebuie să creăm un nod rădăcină, pentru a conține intrarea care distinge între ele cele două pagini de index secționate (de secționare). Arborele obținut după inserarea intrării 8* este prezentat în Fig. 8.

O altă variantă a algoritmului de inserare încearcă să redistribuie intrările unui nod N cu un "frate"(al acestuia), înainte de secționarea nodului. Pe această cale se poate înbunătăți gradul mediu de ocupare a spațiului (de memorie!). În contextul de aici, un "frate"("sibling") al unui nod N este un nod aflat imediat la stânga nodului N şi având acelaşi părinte ca N. Procedura de redistribuire (pe care nu o mai descriem în detaliu) conduce la rezultatul din Fig. 9.

Alte probleme : algoritmul de eliminare cu index cu arbore B+, chei duplicate, arborii B+ în practică - compresia cheilor, procesul "bulk – loading"în arbori B+", conceptul de ordine.

Arborii B+ în sistemele reale: IBM DB2, Informix, Microsoft SQL Server, Oracle, Sybase ASE – toate au index-uri cu arbori B+ "clustered" şi "unclustered".

Folosirea index-urilor în Oracle (SQL – Oracle)

Oracle are 2 metode de a accesa linii într-un tabel: acces secvențial şi acces direct – ca metode.

Acces secvențial – descris cel mai bine ca "răsfoirea unui tabel linie cu linie". Oracle citeşte fiecare linie din tabel. Dacă se caută doar o linie şi tabelul are multe linii – căutarea = ineficientă, timp mult. Este ca şi cum s-ar trece prin o carte de telefon, pagină cu pagină. Dacă noi căutăm numărul de telefon al cuiva al cărui nume începe cu L, nu vom începe căutarea de la A!!

Când Oracle foloseşte metoda de acces direct, citeşte numai liniile care au anumite proprietăți cerute. Pentru aceasta însă este necesar un index.

În Oracle – index-ul – construit ca un arbore, format din noduri – vezi figura:

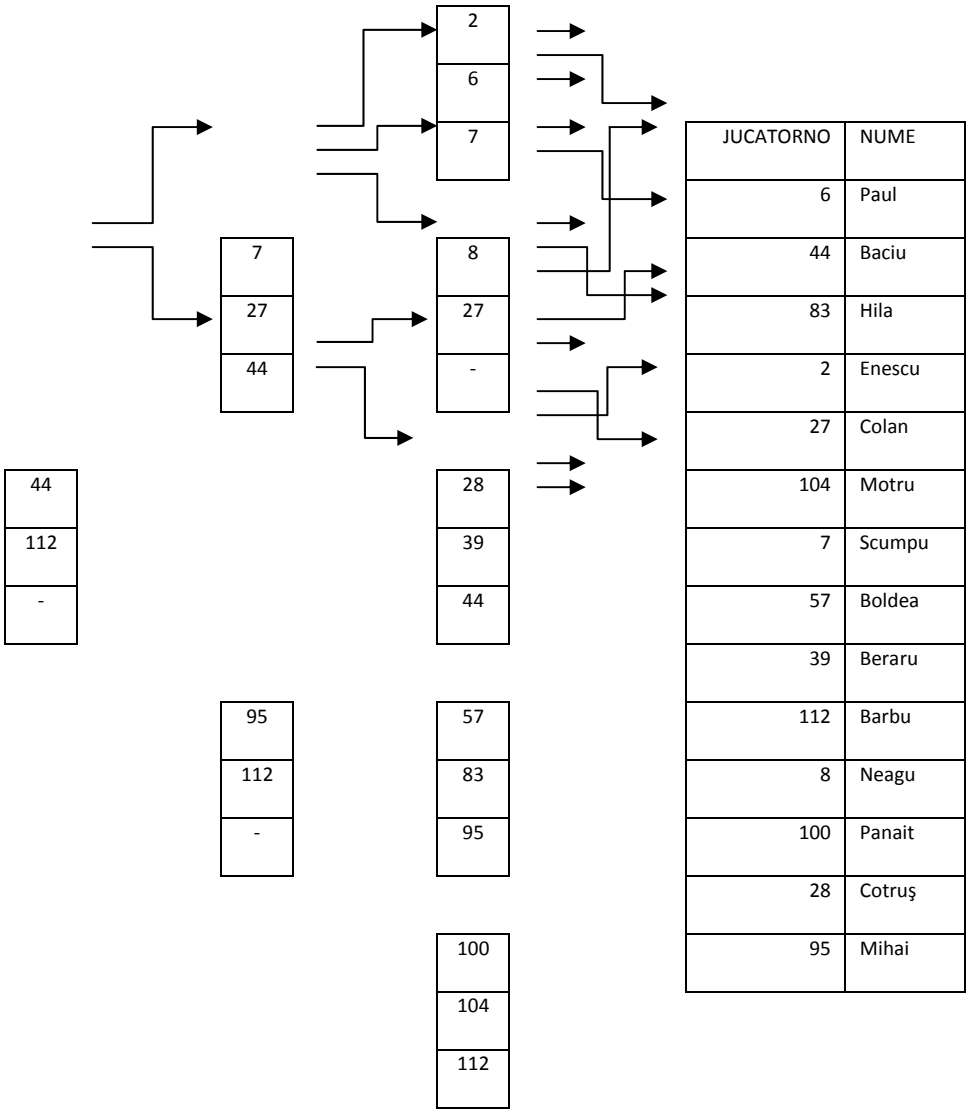


Fig. 10

În stânga – structura de index. În dreapta – două coloane ale unui tabel Jucători (!!!).

Nodul din extrema stânga – rădăcină. Fiecare nod conține un maximum de 3 valori ordonate din coloana JUCATORNO. Fiecare valoare într-un nod indică spre un alt nod sau spre o linie din tabela JUCATORI. (notă - în figură nu am indicat fiecare destinație de “pointer”). De asemenea, observăm că fiecare linie în tabel este “referită” prin cel puțin un nod. Un nod care indică spre o linie este pagină frunză.

Valorile în noduri: ordonate. Pentru fiecare nod, afară de rădăcină, valorile sunt totdeauna <= valoarea care indică spre acel nod. Paginile frunză sunt ele însele legate unele cu altele. O pagină frunză are

un “pointer” spre pagina frunză cu mulțimea vecină de valori (vecină, în sensul de >). Aceşti ultimi pointeri – cu linie groasă.

Oracle – 2 algoritmi pentru lucrul cu index-uri:

- primul – caută liniile în care apare o valoare particulară
- al doilea – răsfoieşste prin o întreagă tabelă sau prin o parte a unei tabele, prin intermediul unei coloane ordonate.

Mai jos – exemple. Primul arată cum Oracle foloseşte un index pentru a selecta o linie (tuplu) particulară.

Ex. 1 – să găsim toate liniile care conțin jucătorul cu no. 44.

Pas 1 – se caută în rădăcina indexului. Rădăcina devine nodul activ.

Pas 2 – Este nodul activ o pagină frunză? Dacă da -> Pasul 4. Dacă nu -> Pasul 3.

Pas 3 – Nodul activ conține 44? Dacă da, nodul către care indică această valoare devine nod activ; apoi - întoarcere la Pas 2.

Dacă nu, alege cea mai mică valoare care este > 44 în nodul activ. Nodul către care indică această valoare devine nod activ; apoi - întoarcere la Pas 2.

Pas 4 – Se caută valoarea 44 în nodul activ. Această valoare indică acum spre toate liniile în tabela JUCĂTORI în care coloana JUCATORNO are valoarea 44. Se extrag toate aceste linii din BD, pentru prelucrare. NOTĂ - putem avea jucători cu acelaşi nume, dar numere diferite.

Astfel, s-au găsit liniile căutate fără a se “răsfoi” prin toate liniile. Timpul de căutare cu index – este <<.

Ex. 2 - extrage toți jucătorii (numele) ordonați după număr.

Pas 1 – Se caută paginile frunză cu vaoloarea cea mai mică. Ea devine nod activ.

Pas 2 – Extrage toate liniile către care indică (“points”) valorile din nodul activ, pentru prelucrare ulterioară eventuală.

Pas 3 – Dacă există o pagină frunză următoare, este făcută nod activ şi se continuă de la Pasul 2.

Etc., Etc….

Crearea index-urilor

Un index se defineşte în Oracle pe o coloană sau pe o combinație de coloane.

Exemplu: CREATE INDEX GOLURI-INDEX ON

REZULTATE (NRGOLURI).

Mai sus, comanda defineşte un index numit GOLURI-INDEX pe coloana NRGOLURI ale unui tabel REZULTATE.

Comanda ajută la o interogare ca:

SELECT * * toate!

FROM REZULTATE

WHERE NRGOLURI = 15

În exemplul de mai sus, Oracle examinează numai liniile din BD care satisfac condiția WHERE - şi deci răspuns rapid. Indexul asigură un acces direct la liniile din tabela inspectată.

NOTĂ - un index nu este niciodată referit explicit în comandă SELECT.

Cheie de căutare ("Search key")

Cheia de căutare este o colecție de câmpuri (unul sau mai multe) ale fişierului de înregistrări pentru care se construieşte un index. Fiecare index are o cheie de căutare asociată. Orice submulțime de cămpuri poate fi o cheie de căutare.

Într-un fişier index, paginile sunt organizate astfel încât să fie posibilă localizarea acelor intrări din index care au o valoare dată a cheii de căutare. Observăm că o cheie de căutare nu trebuie să identifice în mod unic înregistrările.

Alte elemente importante : formatul paginii, formate de înregistrări.