Alen Prodan
Login d.o.o.
Mihačeva draga b.b.
51000 Rijeka
+385 91 156 44 68
alen.prodan@login.hr
http://www.login.hr

ORACLE BALANCED TREE INDEKSI
UNDERSTANDING ORACLE BALANCED TREE INDEXES

SAŽETAK

Balanced tree indeksi su objekti baze podataka čija je osnovna namjena efikasnije pronalaženje redaka u tablicama baze podataka, te ponekad, mogu značajno ubrzati izvođenje SQL naredbi. B-tree indeksi imaju hijerarhijsku strukturu u obliku stabla, a podaci unutar blokova indeksa uvijek su sortirani prema vrijednosti indeks ključa. Održavanje strukture indeksa prilikom modifikacije podataka je resursno intenzivnije u odnosu na regularne heap tablice. Specifični obrasci izmjene podataka, kao i posebnosti u arhitekturi sustava mogu umanjiti efikasnost internih struktura indeksa. Kroz rad će biti prezentirana rješenja kojima je moguće spriječiti ili ukloniti posljedice umanjenja efikasnosti struktura indeksa.

ABSTRACT

B-tree indexes are Oracle schema objects that exists primarily to enhance performance. Effective indexing can result in huge improvements to SQL performance. The advantages of indexes do not come without a cost. The B-tree indexes have a hierarchical tree structure and it takes Oracle more resources to maintain this structure when modifying index blocks compared to regular heap table blocks. Specific data processing patterns can significantly degrade index efficiency and possibly index performance. This paper provides an insight into the way data processing patterns and specific system architecture can affect internal index structures and possibly cause index efficiency problems.

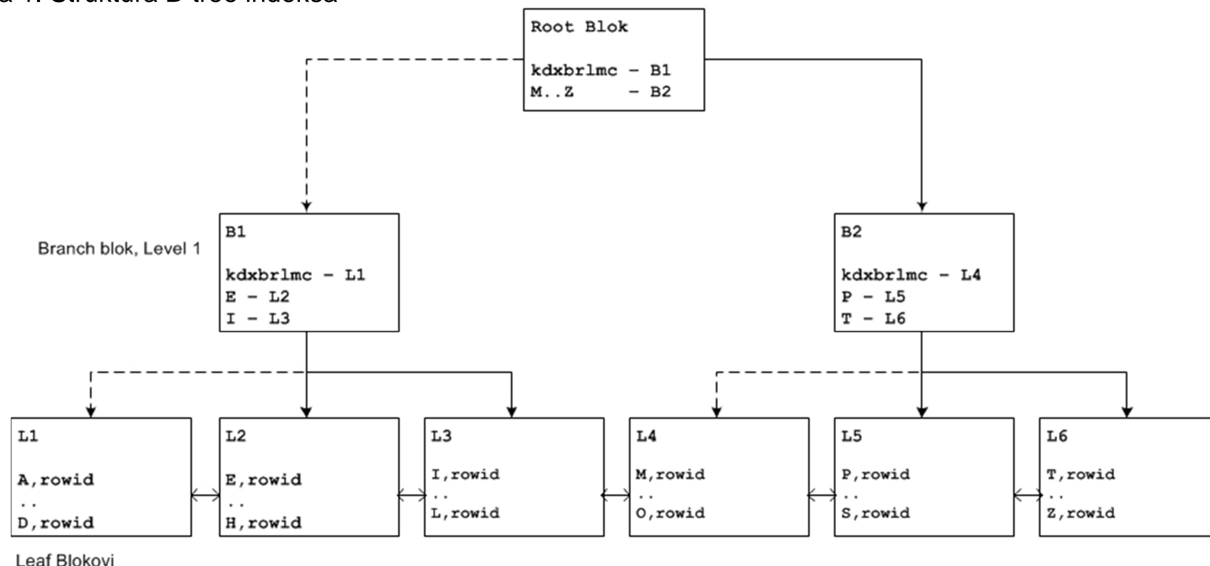
UVOD

Najvažniji cilj fizičkog dizajna baze podataka je minimizirati korištenje I/O resursa. U najvećem broju slučajeva, to se postiže upravo ciljanim kreiranjem indeksa nad jednim ili više stupaca neke tablice. Sa jedne strane, indeksima je moguće reducirati korištenje fizičkih I/O resursa, dok sa druge strane, neprimjereno korištenje indeksa, kao i samo postojanje i korištenje indeksa u specifičnim okolnostima može dramatično povećati korištenje CPU resursa (logički I/O) ili pak izazvati kontenciju među procesima (sesijama) baze podataka. Detaljnije razumijevanje internih struktura indeksa i načina njihovog održavanja pomaže nam u fazi dizajna sustava, ali i kasnijoj optimizaciji i otklanjanju problema u produkcijskoj okolini.

1. INTERNA STRUKTURA B-TREE INDEKSA

Balanced tree indeksi ili skraćeno B-tree indeksi, najčešće su korišteni tip indeksa u oracle bazi podataka. Podaci su u B-tree indeksu uvijek sortirani prema vrijednostima indeks ključa. U strukturi B-tree indeksa, pojavljuju se dvije vrste blokova: branch blokovi služe za pretraživanje (navigaciju) unutar internih struktura indeksa, te leaf blokovi u kojima su pohranjene vrijednosti indeksiranih stupaca tablice (indeks ključ) i adrese redaka u blokovima tablice (rowid). Struktura B-tree indeksa prikazana je na Slici 1.

Slika 1. Struktura B-tree indeksa



Pretraživanje zapisa pohranjenih u strukturi indeksa vrši se navigacijom kroz blokove indeksa, počevši od najviše razine branch blokova prema leaf blokovima. Za razliku od branch blokova, leaf blokovi međusobno su povezani sa susjednim leaf blokovima pokazivačima (pointerima) i to pokazivačem kdxleprv koji sadrži adresu prethodnog (susjednog) leaf bloka, te pokazivača kdxlenxt koji sadrži adresu sljedećeg (susjednog) leaf bloka u strukturi indeksa. Tako povezani leaf blokovi čine double link listu koja omogućuje efikasno pretraživanje ranga vrijednosti putem INDEX RANGE SCAN operacije. Interna struktura root bloka identična je strukturi branch blokova, no kada se indeks sastoji isključivo od samo jednog bloka (root bloka), tada root blok ima internu strukturu leaf bloka.

Osnovna osobina B-tree indeksa je da su oni uvijek uravnoteženi (engl. balanced). Drugim riječima, to znači da je svaki leaf blok uvijek podjednako udaljen od root bloka, te je potrebna približno jednaka količina vremena za čitanje bilo kojeg zapisa, neovisno o tome u kojem se dijelu (leaf bloku) indeksa željeni zapis nalazio. **Visina** (**height**) indeksa označava broj blokova koje je potrebno prijeći od root bloka do leaf bloka. Branch level (blevel) izračunava se kao visina (height) umanjena za jedan (za razinu leaf blokova).

Većina zapisa u branch blokovima su parovi vrijednosti (parcijalni indeks ključ, adresa bloka) i ti su zapisi evidentirani u direktoriju redaka (row directory) branch bloka. Parcijalni zapisi indeks ključeva su minimalna potrebna informacija za usmjeravanje pretrage kroz strukturu indeks stabla, čime se između ostaloga postiže veća gustoća zapisa u branch blokovima u usporedbi sa leaf blokovima, te dobiva na uštedi diskovnog prostora. Nadalje, kako bi se maksimizirao broj zapisa u branch blokovima, oracle će ignorirati PCTFREE postavku prilikom kreiranja ili rekreiranja (rebuild) indeksa, ali isključivo za branch blokove, dok će leaf blokovi poštovati PCTFREE postavku.

Detaljniji uvid u strukturu indeks stabla može se dobiti koristeći treedump dijagnostički event, za što nam je potreban user_objects.object_id indeksa kojega želimo analizirati:

```
select object_id, object_type from user_objects where object_name = 'T5_DATUM' ;

 OBJECT_ID OBJECT_TYPE
---------- --------------------
     73444 INDEX
```

Usporedbe radi, prije samog generiranja treedump eventa, prikupiti ćemo osnovne statistike o indeksu putem analyze index naredbe, kako bi ih mogli usporediti sa rezultatom treedump naredbe:

```
analyze index t5_datum validate structure;

select br_blks, lf_blks, br_rows, lf_rows, pct_used from index_stats;

   BR_BLKS    LF_BLKS    BR_ROWS    LF_ROWS   PCT_USED
---------- ---------- ---------- ---------- ----------
         1         10          9       3650         80
```

Generirani rezultat treedump naredbe možemo pronaći u trace datoteci serverskog procesa:

```
alter session set events 'immediate trace name treedump level 73444';

----- begin tree dump
branch: 0x1004fe1 16797665 (0: nrow: 10, level: 1)
   leaf: 0x1004fe2 16797666 (-1: nrow: 377 rrow: 377)
   leaf: 0x1004fe3 16797667 (0: nrow: 377 rrow: 377)
   leaf: 0x1004fe4 16797668 (1: nrow: 377 rrow: 377)
   leaf: 0x1004fe5 16797669 (2: nrow: 377 rrow: 377)
   leaf: 0x1004fe6 16797670 (3: nrow: 377 rrow: 377)
   leaf: 0x1004fe7 16797671 (4: nrow: 377 rrow: 377)
   leaf: 0x1004fe8 16797672 (5: nrow: 377 rrow: 377)
   leaf: 0x1004fe9 16797673 (6: nrow: 377 rrow: 377)
   leaf: 0x1004fea 16797674 (7: nrow: 377 rrow: 377)
   leaf: 0x1004feb 16797675 (8: nrow: 257 rrow: 257)
----- end tree dump
```

Treedump će generirati po jedan redak u trace datoteci za svaki blok u indeksu. Prvi redak treedump rezultata pokazuje na first level branch block (level: 1). Branch blok sadrži pokazivače na 10 leaf blokova (nrow: 10). Ostali retci prikazuju informacije o leaf blokovima. Budući da su svi indeks leaf blokovi na razini 0, treedump prikaz za leaf blokove ne sadrži podatak o levelu, ali sadrži dva podatka specifična za leaf blokove: rrow predstavlja trenutni broj redaka u leaf bloku, dok nrow predstavlja broj elemenata u row directoryiju pojedinog leaf bloka. Kada je rrow manji od nrow to nam ukazuje da postoje izbrisani indeks retci koji nisu još počišćeni. Prilikom prvog sljedećeg inserta u taj indeks blok ili posredstvom delayed-block cleanout mehanizma izbrisani indeks retci se brišu iz row directoryija, te rrow veličina postaje jednaka nrow veličini.

U svakom retku nalazi se adresa bloka (DBA) u heksadecimalnom i u dekadskom obliku. Zanimljivo je da prvi leaf blok na kojega pokazuje root blok ima redni broj -1. Netipična numeracija ukazuje da je riječ o specifičnoj optimizaciji, a naznaka njezina postojanja vidljiva je i na Slici 1. ako pratimo tijek isprekidanih linija pokazivača sa branch level 2 bloka (root) prema branch level 1 blokovima, te sa branch level 1 blokova prema leaf blokovima. Naime, osim ranije spomenutih zapisa pohranjenih u row directoryiju branch blokova u formatu (parcijalni indeks ključ, adresa bloka), postoji još jedan posebni zapis koji se zove "leftmost child", a u simboličkom dumpu branch bloka vidimo ga pod oznakom "kdxbrlmc". Leftmost child zapis sadrži samo adresu branch ili leaf bloka. Parcijalni indeks ključ koji je standarno pridružen zapisima u branch blokovima svjesno je izostavljen iz row directoryija kako bi se uštedjelo na diskovnom prostoru u branch blokovima budući da proces koji pretražuje indeks može izvesti vrijednosti koje pokriva leftmost child blok na temelju vrijednosti sadržanih u njegovom nadređenom (parent) bloku unutar hijerarhije indeksa. Na primijeru sa Slike 1., pretragom želimo pronaći retke koji u indeks ključu imaju vrijednost 'A'. Iz root bloka dobivamo informaciju da branch blok B2 sadrži podatke od M do Z. Dakle, traženi podatak ukoliko postoji mora biti u bloku B1. U bloku B1 pronalazimo informaciju da leaf blokovi L2, L3, itd., imaju podatke od slova E, iz čega proizlazi da se željeni zapisi moraju nalaziti u L1 leaf bloku. Na poslijetku, u L1 leaf bloku pronalazimo zapise sa indeks ključem "A", te preko rowid vrijednosti pronalazimo retke u blokovima tablice nad kojom je indeks kreiran.

Konkretno, informaciju o leftmost child zapisu, ali i pregršt ostalih informacija o pojedinom indeks bloku možemo pronaći u simboličkom prikazu (symbolic dump) root branch bloka, kojega generiramo na sljedeći način:

```
select
dbms_utility.data_block_address_file(16797665)||'/'||
dbms_utility.data_block_address_block(16797665) fno_blk from dual;

FNO_BLK
--------------------------------------------------------------------------------
4/20449

alter system dump datafile 4 block 20449;
```

Isječak iz simboličkog pregleda root bloka:

```
Block header dump:  0x01004fe1
 Object id on Block? Y
 seg/obj: 0x11ee4  csc: 0x00.1dcd29  itc: 1  flg: -  typ: 2 - INDEX
     fsl: 0  fnx: 0x0 ver: 0x01
```

```
 Itl           Xid                  Uba          Flag  Lck        Scn/Fsc
0x01   0xffff.000.00000000  0x00000000.0000.00  C---    0  scn 0x0000.001dcd29
Branch block dump
=================
header address 47010060950084=0x2ac161fbda44
kdxcolev 1
KDXCOLEV Flags = - - -
kdxcolok 0
kdxcoopc 0x80: opcode=0: iot flags=--- is converted=Y
kdxconco 2
kdxcosdc 0
kdxconro 9
kdxcofbo 46=0x2e
kdxcofeo 7885=0x1ecd
kdxcoavs 7839
kdxbrlmc 16797666=0x1004fe2 /* pokazivač na leftmost child blok (-1) */
kdxbrsno 0
kdxbrbksz 8056
kdxbr2urrc 0
row#0[8037] dba: 16797667=0x1004fe3  /* pokazivač na drugi leaf blok (0) */
col 0; len 7; (7):  78 70 01 02 01 01 01
col 1; len 6; (6):  01 00 4f d1 01 79
row#1[8018] dba: 16797668=0x1004fe4 /* pokazivač na treći leaf blok (1) */
col 0; len 7; (7):  78 70 01 03 01 01 01
col 1; len 6; (6):  01 00 4f d2 01 41
....
```

U nastavku slijedi tumačenje pojedinih zapisa iz simboličkog pregleda branch bloka:

- **itc: 1** - broj slotova u (ITL) interested transaction listi. Branch blokovi imaju standarno jedan ITL slot kojega koriste isključivo za blok split operacije, dok leaf blokovi imaju minimalno 2 slota.
- **kdxcolev 1** ukazuje da blok pripada prvoj razini (level 1), što potvrđuje da se nalazi u branch razini, budući da level 0 blokovi pripadaju leaf razini.
- **kdxconco 2** označava da indeks ima dva stupca. **kdxconro 9** označava da root blok ima 9 redaka u row directoryiju, što potvrđuje da se leftmost child ne pohranjuje u row directoryiju (leaf blok numeriran sa -1).
- **kdxcofbo 46=0x2e** je adresa unutar bloka na kojoj počinje raspoloživ heap prostor za pohranu redaka, dok **kdxcofeo 7885=0x1ecd** označava adresu na kojoj završava trenutno raspoloživ heap prostor. **kdxcoavs 7839** označava količinu raspoloživog heap prostora unutar bloka, efektivno to je **kdxcofeo-kdxcofbo**.
- **kdxbrlmc** je pointer na adresu leftmost child branch ili leaf bloka.
- **kdxbrbksz** je ukupno raspoloživi prostor u bloku.
- linije u traceu koje počinju sa **row#0[8037] dba: 16797667=0x1004fe3** prikazuju adresu i indeks ključ drugog leaf bloka, **row#1[8018]** trećeg leaf bloka, itd.

Najznačajnija razlika Block headera između branch i leaf blokova je u broju ITL slotova (itc). Naime leaf indeks blokovi imaju po default postavkama 2 ITL elementa. Prvi ITL slot rezerviran je za operacije dijeljenja blokova (block split), dok se ITL slotovi na poziciji 2 (ili više) koriste za transakcije nad podacima:

```
Block header dump:  0x01004fe2
 Object id on Block? Y
 seg/obj: 0x11eea  csc: 0x00.1e099e  itc: 2  flg: -  typ: 2 - INDEX
     fsl: 0  fnx: 0x0 ver: 0x01

 Itl           Xid                  Uba          Flag  Lck        Scn/Fsc
0x01   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x02   0xffff.000.00000000  0x00000000.0000.00  C---    0  scn 0x0000.001e099e
```

Nadalje, za razliku od branch blokova, indeks header sekcija leaf blokova sadrži dva dodatna elementa kdxleprv i kdxlenxt koji predstavljaju pokazivače (pointere) prema susjednim leaf blokovima, čime tvore double link listu kojom se omogućuje efikasno pretraživanje kod index range scan operacija. U konkretnom slučaju kdxleprv=0, što znači da je riječ o prvom leaf bloku:

```
Leaf block dump
===============
header address 47920845953628=0x2b9571001a5c
kdxcolev 0
KDXCOLEV Flags = - - -
kdxcolok 0
kdxcoopc 0x80: opcode=0: iot flags=--- is converted=Y
kdxconco 1
kdxcosdc 0
kdxconro 398
kdxcofbo 832=0x340
kdxcofeo 1664=0x680
kdxcoavs 832
kdxlespl 0
kdxlende 0
kdxlenxt 16797667=0x1004fe3
kdxleprv 0=0x0
kdxledsz 6
kdxlebksz 8032
```

U leaf blokovima postoje dvije varijacije zapisa koji sadrže indeks ključeve. Kod non-unique indeksa simbolički prikaz zapisa izgleda kao što slijedi:

```
row#0[8015] flag: ------, lock: 0, len=17
col 0; len 7; (7):  78 70 01 01 01 01 01 /* indeksirani stupac tablice, datumsko
                                           polje, 7-byteova */
col 1; len 6; (6):  01 00 4f d1 00 00    /* rowid, automatski postaje sastavni dio
                                           index ključa i garantira uniqueness */
```

Za primjetiti je da u internoj implementaciji non-unique indeksa, rowid adresa retka u tablici automatski postaje drugi element indeks ključa, što ima dvije važne implikacije: prvo, kombinacija (col0=datum, col1=rowid) garantira jednoznačnost (uniqueness) svakog indeks retka; drugo, rowid vrijednost sudjeluje u redoslijedu sortiranja indeks redaka unutar indeks strukture, što može imati kasnije implikacije prilikom DML operacija nad podacima.

Kod unique b-tree indeksa simbolički prikaz izgleda kao što slijedi:

```
row#0[8016] flag: ------, lock: 0, len=16, data:(6):  01 00 4f d1 00 00  /* rowid
col 0; len 7; (7):  78 70 01 02 01 01 01 /* indeksirani stupac tablice, 7-byte date
```

Kod unique b-tree indeksa, rowid i dalje postoji kao podatak u indeks retku (data:(6): 01 00 4f d1 00 00), međutim rowid više nije sastavni dio indeks ključa, te je vidljivo da indeks ključ ima samo jedan element (col 0). To znači da se jednoznačnost (uniqueness) retka u strukturi indeksa određuje isključivo na temelju vrijednosti datumskog polja nad indeksiranim stupcem tablice.

Indeksi se automatski ažuriraju u sklopu transakcija kojima se mijenjaju podaci u tablicama nad kojima su indeksi kreirani. Te izmjene mijenjaju sadržajno zapise pohranjene u blokovima indeksa, a posljedično utječu i na izmjenu internih fizičkih struktura indeksa. Interna struktura indeks blokova dijeli mnoge zajedničke elemente sa internom strukturom blokova najčešće korištene vrste tablica - heap tablica. Prilikom unosa novog retka u tablicu, on se zapisuje pri dnu heap prostora u jednom od blokova tablice sa dovoljno slobodnog prostora. Novo-insertirani redak dobivaj svoj element (slot) unutar row directoryija bloka tablice, gdje mu se pridjeljuje redni broj i evidentira adresa (offset) na kojoj se nalazi početak retka u heap prostoru. Redni broj retka javno je eksponirani podatak vidljiv kao zadnji (row) element rowid adrese (object/file/block/row). Memorijska adresa (offset) retka unutar bloka tablice nije eksponirana, te omogućuje kernel kodu transparentno preslagivanje redaka unutar bloka tablice kako bi se optimiziralo korištenje prostora unutar bloka, npr. u slučaju fragmentacije prostora. Kod indeks blokova, retci (indeks ključevi+rowid), također se pohranjuju pri dnu slobodnog heap prostora unutar bloka indeksa, ali su za razliku od tablica elementi unutar row directoryija sortirani redoslijedno prema vrijednosti indeks ključa. U row directory nisu pohranjene vrijednosti indeks ključeva već samo memorijske adrese na kojima se nalaze indeks ključevi, unutar heap prostora indeks bloka.

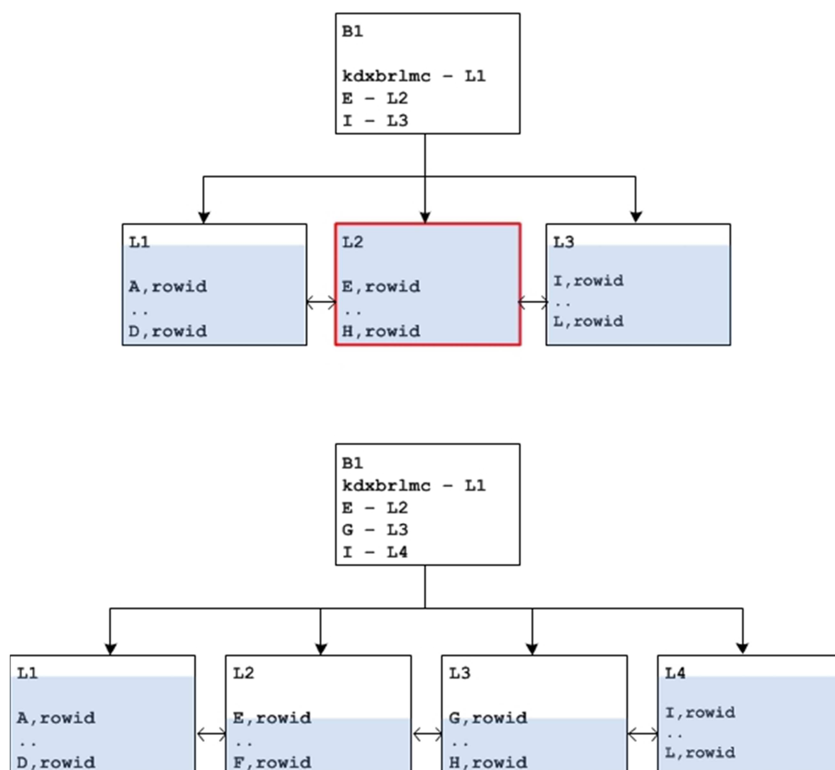
Činjenica da je row directory unutar svakog indeks bloka sortiran prema vrijednostima indeks ključa omogućuje pretraživanje indeks ključeva pohranjenih u indeks blokove putem algoritma za binarno pretraživanje (binary search) i to kako kod pretrage pojedinačnih vrijednosti, tako i kod

pretrage ranga vrijednosti gdje su tražene vrijednosti >=min(vrijednosti) i <=max(vrijednosti). Binary search algoritam radi nad sortiranom matricom (array) dijeleći u svakoj iteraciji na pola skup elemenata kojega pretražuje, pa vrijeme odziva pretraživanja elemenata unutar matrice raste logaritamskom skalom koja se izražava kao O (log (2,N)). To znači da kod eksponencijalnog rasta količine redaka koje pretražujemo, vrijeme odziva raste linearno. Pretpostavimo da je za pretragu matrice sa N=2 elementa potrebna 1 milisekunda, tada bi za pretragu N=4 bilo potrebno log(2, 4) = 2 ms, za N=8 potrebno je log(2,8) = 3 ms, itd. Budući da su indeks leaf blokovi prilično gusto "naseljeni", te da nije neuobičajeno da leaf blokovi sadrže i po nekoliko stotina redaka, mogućnost efikasnog lociranja retka je vrlo važna. Npr., indeks nad stupcem datumskog tipa sa 8 KB veličinom bloka, uz standardni PCTFREE 10%, može sadržavati oko 377 indeks ključeva. Algoritmom binarnog pretraživanja potrebno je posjetiti maksimalno log (2, 377) = 9 elemenata kako bi se pronašla tražena vrijednost. Pod pretpostavkom da je vrijeme odziva (resursi) potrebni za posjetu svakom retku unutar bloka fiksno, to čini tek 9/377 * 100 = 2,38% resursa potrebnih u odnosu na sekvencijalno O (N) pretraživanje kada vrijednosti ne bi bile sortirane, te bi bilo potrebno posjetiti svaki od 377 redataka kako bi provjerili zadovoljava li uvjet pretraživanja.

1.1. Dioba indeks blokova putem block split mehanizma

Prilikom kreiranja ili rekreiranja (rebuild) indeksa, u svakom leaf bloku indeksa ostaje rezerviran slobodni prostor u iznosu određenom PCTFREE atributom. Ako nije drugačije određeno, osnovna (default) vrijednost PCTFREE atributa iznosi 10%, što znači da svaki leaf blok nakon kreiranja sadrži 10% slobodnog prostora. Taj slobodan prostor smanjuje incidenciju naknadne diobe blokova (index block split) - standarnog mehanizma koji se koristi za dodjelu novih blokova strukturi indeksa. U slučaju nedostatka prostora u tekućem bloku, javlja se potreba za alokacijom dodatnog prostora kroz mehanizam diobe blokova. Operacije diobe blokova (block split) su resursno relativno zahtjevne operacije i ako je moguće, bolje ih je izbjegavati. Dva su osnovna modaliteta diobe blokova: 50-50 block split i 90-10 block split. Oba modaliteta mogu se primjeniti na root, branch i leaf blokove.

Slika 2. Primjer diobe blokova 50-50 block split mehanizmom

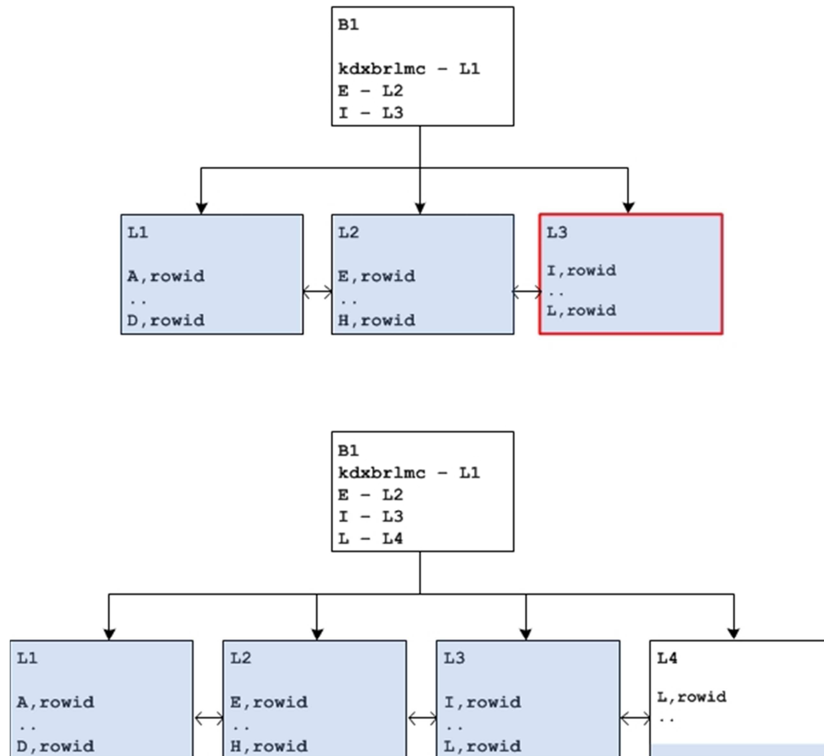


Konceptualno 50-50 block split izvodi se na sljedeći način:

1. Indeksu se dodjeljuje novi blok, preuzet sa freeliste

2. Preraspodijeljuju se podaci u bloku koji se dijeli, na način da doljnja polovica (po volumenu) indeks redaka ostane u postojećem bloku, dok se gornja polovica kopira u novi blok
3. Novi indeks redak pohranjuje se u odgovarajući leaf blok
4. kdxlenxt pokazivač originalnog bloka pokazuje na novi blok
5. kdxleprv pokazivač desnog susjeda originalnog bloka također pokazuje na novi blok
6. Ažurira se branch blok koji referencira originalni blok, te se dodaje novi zapis koji će referencirati najmanju vrijednost indeks ključa u novom leaf bloku
7. Ako u branch bloku iz koraka 6. nema dovoljno prostora, tada se izvršava 50-50 branch blok split dioba branch bloka

Slika 3. Primjer diobe blokova 90-10 block split mehanizmom



Konceptualno 90-10 blok split izvodi se na sljedeći način:

1. Ukoliko je novi indeks redak jednak ili veći od trenutne maksimalne vrijednosti indeks ključa, tada se izvodi 90-10 blok split dioba
2. 90-10 indeks blok split je tipičan scenario koji spriječava fragmentiranje prostora u indeks blokovima u koji se pohranjuju vrijednosti generirane putem oracle sequencea ili sličnog mehanizma koji generira uvijek rastući niz vrijednosti
3. Naziv 90-10 blok split nije sasvim korektan, budući da se u novi blok prenosi manje od 10% podataka (bliži je iznos od 1%).

Sustav će primjeniti 90-10 blok split diobu samo ukoliko je vrijednost koja se upisuje jednaka ili veća od trenutne maksimalne vrijednosti indeks ključa. U svim ostalim slučajevima koristiti će se 50-50 blok split mehanizam.

2. UTJECAJ OBRAZACA IZMJENE PODATAKA NA INTERNE STRUKTURE B-TREE INDEKSA

Upravljajnje prostorom ovisno je o vrsti DML naredbi i uzorcima mijenjanja podataka kao i razini konkurentnosti, pa ćemo pokušati utvrditi utjecaj insert,delete i update DML naredbi na upravljanje prostorom u blokovima indeksa razrađujući neke od tipičnih scenarija u primjerima koji slijede.

Primjer 1.: brisanje indeks blokova u lijevom dijelu indeksa, te insert novih podataka u desnom dijelu indeksa

Potpuno brisanje svih redaka iz blokova indeksa putem DELETE naredbe generalno ne predstavlja problem za funkcioniranje indeksa, kao što možemo vidjeti na sljedećem primjeru na kojem ćemo simulirati brisanje povijesnih podataka iz tablice, te učitavanje novih podataka. Test započinjemo pripremom tablice u koju učitavamo inicijalni skup podataka od 2000 redaka, uz kreiranje indeksa T_DATUM nad stupcem datum:

```
create table t
(
 id number,
 datum date
);

insert into t
select rownum, trunc(sysdate)+trunc((rownum-1)/200)
from dual
connect by level <= 2000;

commit;

create index t_datum on t(datum);
```

Uz pomoć interne sys_op_lbid funkcije dolazimo do rowid adrese indeks blokova u koje su pohranjeni podaci, te u konačnici putem dbms_rowid.rowid_block_number funkcije dolazimo do broja bloka u numeričkom formatu:

```
select object_id from user_objects where object_name = 'T_DATUM';
73482 <= ovo je object_id

select dbms_rowid.rowid_block_number(
sys_op_lbid(73482, 'L' , t.rowid)) blknum, min(datum), max(datum), count(*)
from t
group by dbms_rowid.rowid_block_number(
sys_op_lbid(73482, 'L' , t.rowid))
order by 1;


    BLKNUM MIN(DATU MAX(DATU   COUNT(*)
---------- -------- -------- ----------
     20802 24.08.12 25.08.12        377
     20803 25.08.12 27.08.12        377
     20804 27.08.12 29.08.12        377
     20805 29.08.12 31.08.12        377
     20806 31.08.12 02.09.12        377
     20807 02.09.12 02.09.12        115

6 rows selected.
```

Iz prethodnog upita vidljivo je da indeks ima 6 leaf blokova. Kao rezultat brisanja svih redaka sa datumom manjim od 02.09.2012 trebali bi imati 4 potpuna prazna leaf bloka (blokovi 20802 do 20805):

```
delete from t where datum < to_date('02.09.2012', 'dd.mm.yyyy');
commit;

exec dbms_stats.gather_table_stats(ownname=>user, tabname=>'T', cascade=>true);
```

Specifičnost b-tree indeksa je da se potpuno prazni indeks leaf blokovi koji su sadržavali izbrisane retke istovremeno nalaze na freelisti, ali još uvijek čine i sastavni dio strukture T_DATUM indeksa, što možemo provjeriti jednostavnim SQL upitom koji generira INDEX FULL SCAN operaciju koja će posjetiti ukupno 6 blokova (obratiti pažnju na consistent gets statistiku):

```
select min(datum) from t;

MIN(DATU
--------
02.09.12


Execution Plan
----------------------------------------------------------
Plan hash value: 381944028

---------------------------------------------------------------------------
| Id  | Operation                   | Name     | Rows  | Bytes | Cost (%CPU)|
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |          |     1 |     8 |     2   (0)|
|   1 |  SORT AGGREGATE             |          |     1 |     8 |            |
|   2 |   INDEX FULL SCAN (MIN/MAX)| T_DATUM  |     1 |     8 |     2   (0)|
---------------------------------------------------------------------------


Statistics
----------------------------------------------------------
             ...
          6  consistent gets   <= SQL query je posjetio svih 6 leaf blokova
          0  physical reads
             ...
          1  rows processed
```

Zatim ćemo simulirati unos novih redaka u tablicu, na način da ćemo ponovno insertirati 1800 redaka koji će imati vrijednost datuma veću od 02.09.2012, kako bi ustanovili da li će oracle reciklirati već postojeće prazne blokove, koji se ujedno nalaze i na freelisti:

```
insert into t
select rownum+2000, to_date('03.09.2012', 'dd.mm.yyyy')+trunc((rownum-1)/200)
from dual
connect by level <= 1800; commit;
```

Iz priloženog treedump rezultata vidljivo je da su svi stari blokovi iskorišteni, te da sada ponovno sadržavaju i do 420 novih redaka:

```
*** 2012-08-24 12:43:28.460
----- begin tree dump
branch: 0x1005141 16798017 (0: nrow: 7, level: 1)
   leaf: 0x1005146 16798022 (-1: nrow: 85 rrow: 85)
   leaf: 0x1005147 16798023 (0: nrow: 215 rrow: 215)
   leaf: 0x1005145 16798021 (1: nrow: 420 rrow: 420)
   leaf: 0x1005144 16798020 (2: nrow: 420 rrow: 420)
   leaf: 0x1005143 16798019 (3: nrow: 420 rrow: 420)
   leaf: 0x1005142 16798018 (4: nrow: 420 rrow: 420)
   leaf: 0x1005148 16798024 (5: nrow: 20 rrow: 20)
----- end tree dump
```

Primjer 2.: Fragmentacija prostora u indeks blokovima nastala ažuriranjem većeg broja identičnih, već postojećih indeks zapisa, uz konstantni unos novih redaka u desnom dijelu indeksa.

Za razliku od prethodnog primjera, u kojem je potpuno brisanje indeks blokova na lijevoj strani indeksa uz kontinuirano insertiranje novih slogova sa neprestano rastućim vrijednostima indeks ključa na desnoj strani indeksa omogućilo savršeno recikliranje praznih (izbrisanih) blokova, u sljedećem primjeru biti će prikazan uzorak modifikacije podataka putem UPDATE naredbe koji iza sebe ostavlja veći broj polupraznih indeks blokova zbog diobe blokova putem 50-50 blok split mehanizma. Primjer započinjemo sa tablicom T i indeksom T_DATUM, inicijaliziranima kao u prethodnom primjeru za delete+insert scenario izmjene podataka.

Pretpostavimo da se u doljnu tablicu svaki dan insertiraju novi dokumenti koje zatim korisnici obrađuju, te po završetku obrade potvrđuju dokumenat. Svaki puta prilikom izmjene dokumenta,

ažurira se indeksirano datumsko polje koje označava datum zadnje izmjene dokumenta. Inicijalni set podataka je kao što slijedi:

```
select dbms_rowid.rowid_block_number(
sys_op_lbid(&objid, 'L' , t.rowid)) blknum, min(datum), max(datum), count(*)
from t
group by dbms_rowid.rowid_block_number(
sys_op_lbid(&objid, 'L' , t.rowid))
order by 1;

    BLKNUM MIN(DATU MAX(DATU   COUNT(*)
---------- -------- -------- ----------
     20802 24.08.12 25.08.12        377
     20803 25.08.12 27.08.12        377
     20804 27.08.12 29.08.12        377
     20805 29.08.12 31.08.12        377
     20806 31.08.12 02.09.12        377
     20807 02.09.12 02.09.12        115
```

Najnoviji uneseni dokumenti imaju zadnji datum izmjene 02.09.2012 i nalaze se u indeks bloku 20807. Tijekom dana korisnici obrađuju dokumente unesene između 25.08.2012 i 29.08.2012, te ih nakon obrade potvrđuju čime se i njihova vrijednost datuma zadnje izmjene mijenja na 02.09.2012. Nakon updatea, imamo sljedeću strukturu indeks blokova vidljivu u rezultatu treedump naredbe:

```
----- begin tree dump
branch: 0x1005141 16798017 (0: nrow: 11, level: 1)
   leaf: 0x1005142 16798018 20802 (-1: nrow: 377 rrow: 200)
   leaf: 0x1005143 16798019 20803 (0: nrow: 377 rrow: 0)
   leaf: 0x1005144 16798020 20804 (1: nrow: 377 rrow: 0)
   leaf: 0x1005145 16798021 20805 (2: nrow: 377 rrow: 308)
   leaf: 0x1005146 16798022 20806 (3: nrow: 215 rrow: 215)
   leaf: 0x1005148 16798024 20808 (4: nrow: 215 rrow: 215)
   leaf: 0x1005149 16798025 20809 (5: nrow: 215 rrow: 215)
   leaf: 0x100514a 16798026 20810 (6: nrow: 215 rrow: 215)
   leaf: 0x100514b 16798027 20811 (7: nrow: 215 rrow: 215)
   leaf: 0x100514c 16798028 20812 (8: nrow: 302 rrow: 302)
   leaf: 0x1005147 16798023 20807 (9: nrow: 115 rrow: 115)
----- end tree dump
```

Na temelju prikaza rezultata treedump naredbe, možemo vidjeti efekt UPDATE narebe na internu strukturu b-tree indeksa. Update nareba za indeks leaf blokove, procesira se kao delete+insert, zbog čega retci migriraju iz jednog u drugi leaf blok, budući da indeks ključevi uvijek moraju biti pohranjeni u sortiranom rasporedu, što znači da moraju biti pohranjeni u onaj blok kojem pripadaju na temelju vrijednosti indeks ključa i primjenjenog sort algoritma. Delete faza update naredbe rezultirala je potpunim brisanjem blokova 20803 i 20804, te djelomičnim brisanjem blokova 20802 i 20805. Nakon delete faze slijedi insert faza u indeks blok u kojega redak mora pripasti na temelju nove vrijednosti indeks ključa - u našem slučaju nova vrijednost indeks ključa je 02.09.2012. Iako na prvi pogled zbunjujuće, insert faza update naredbe nije pohranila ažurirane indeks ključeve u blok 20807 koji je u trenutku updatea bio tek 1/3 popunjen a i sadržavao je retke za 02.09.2012., već su se zapisi počeli pohranjivati u blok 20806 koji je već bio 90% popunjen, što je vrlo brzo rezultiralo nizom od nekoliko 50-50 block split operacija, koje iza sebe ostavljaju poluprazne indeks blokove. No zašto je baš izabran blok 20806 ? Odgovor pronalazimo u činjenici da je T_DATUM non-unique indeks, te da se indeks ključ u tom slučaju sastoji od dva elementa (indeks ključ, rowid), te oba sudjeluju u pozicioniranju retka unutar strukture indeksa na temelju primjenjenog sort algoritma. Budući da prva tri elementa rowid adrese označavaju objekt/datoteku/blok, te da se u konkretnom slučaju svi alocirani blokovi nalaze u istoj datoteci, to znači da će ranije insertirani retci imati manji block id u odnosu na kasnije insertirane retke, te posljedično za istu vrijednost prvog dijela indeks ključa (datuma), biti niže rangirani u sort algoritmu. Iz tog razloga re-insertirani retci su pohranjeni u blok 20806.

Primjer 3.: Utjecaj redoslijeda dolaska podataka na fragmentaciju prostora unutar indeks blokova

U sljedećem primjeru ispitati ćemo postoji li utjecaj redoslijeda dolaska podataka u tablicu na fragmentaciju prostora unutar indeks blokova. Započinjemo sa pripremom tablice, indeksa i učitavanjem podataka:

```
create table t
(
 id number,
 d1 date,
 d2 date
);

create index t_d1 on t(d1);
create index t_d2 on t(d2);

insert into t
select rownum,
       to_date('01.01.2012', 'dd.mm.yyyy')+mod(rownum, 10) d1,
       to_date('01.01.2012', 'dd.mm.yyyy')+trunc((rownum-1)/100) d2
from dual
connect by level <=10000; commit;
```

Tablica u primjeru sadrži ukupno 10000 redaka. Stupci D1 i D2 u logičkom smislu sadrže identične podatke: svaka od 10 distinct datumskih vrijednosti ponavlja se 1000 puta. U stupcu D1 vrijednosti se neprestano izmjenjuju, dok stupac D2 ima monotoni rast.

Iz INDEX_STATS viewa, možemo vidjeti osnovne statistike oba indeksa generirane analyze index naredbom:

```
select name, br_blks, lf_blks, br_rows, lf_rows, height, btree_space, pct_used from
index_stats;

NAME      BR_BLKS    LF_BLKS    BR_ROWS    LF_ROWS     HEIGHT BTREE_SPACE   PCT_USED
----- ---------- ---------- ---------- ---------- ---------- ----------- ----------
T_D1           1         39         38      10000          2      319872         60

NAME      BR_BLKS    LF_BLKS    BR_ROWS    LF_ROWS     HEIGHT BTREE_SPACE   PCT_USED
----- ---------- ---------- ---------- ---------- ---------- ----------- ----------
T_D2           1         24         23      10000          2      199932         96
```

Indeks T_D1, kreiran nad stupcem sa cikličkim izmjenama dolaznih vrijednosti ima gotovo 62% više leaf blokova za isti broj indeks redaka, u odnosu na T_D2 indeks kojega je karakterizirao monotoni rast vrijednosti indeks ključa prilikom inserta podataka. Značajan podatak je i prosječna razina iskorištenosti prostora u indeks blokovima, koja kod T_D1 indeksa iznosi relativno niskih 60%, dok kog T_D2 indeksa iznosi odličnih 96%. Letimičan pogled na rezultat treedump naredbe za T_D1 indeks, upućuje na to da je kontinuirano rastao uz 50-50 blok splitove ostavljajući iza sebe poluprazne indeks leaf blokove:

```
----- begin tree dump T_D1
branch: 0x1005141 16798017 (0: nrow: 39, level: 1)
   leaf: 0x1005142 16798018 (-1: nrow: 215 rrow: 215)
   leaf: 0x1005168 16798056 (0: nrow: 215 rrow: 215)
   leaf: 0x1005179 16798073 (1: nrow: 208 rrow: 208)
   ....
```

Suprotno, kod T_D2 indeksa, monotoni rast indeks ključa omogućio je korištenje 90-10 blok split mehanizma i odličnu popunjenost leaf blokova sa 420 indeks zapisa u svakom leaf bloku:

```
----- begin tree dump T_D2
branch: 0x1005149 16798025 (0: nrow: 24, level: 1)
   leaf: 0x100514a 16798026 (-1: nrow: 420 rrow: 420)
   leaf: 0x100514b 16798027 (0: nrow: 420 rrow: 420)
   leaf: 0x100514c 16798028 (1: nrow: 420 rrow: 420)
....
```

Primjer 4.: Utjecaj PCTFREE atributa na razinu iskorištenosti prostora indeks bloka

U sljedećem vrlo jednostavnom primjeru, biti će prikazan loš utjecaj neprimjerenog odabira PCTFREE atributa na razinu iskorištenosti slobodnog prostora u indeks blokovima. Primjer započinjemo pripremom tablice i učitavanjem podataka, te kreiranjem indeksa:

```
create table t
(
 n1 number,
 d1 date,
 d2 date
);

insert into t
select rownum, trunc(sysdate, 'yyyy')+rownum d1, trunc(sysdate, 'yyyy')+rownum d2
from dual
connect by level <= 3650;

commit;

create index t_d1 on t(d1);

create index t_d2 on t(d2) pctfree 50;

analyze index t_d1 validate structure;

select height, br_blks, lf_blks, br_rows, lf_rows from index_stats;

    HEIGHT    BR_BLKS    LF_BLKS    BR_ROWS    LF_ROWS
---------- ---------- ---------- ---------- ----------
         2          1         10          9       3650

analyze index t_d2 validate structure;

select height, br_blks, lf_blks, br_rows, lf_rows from index_stats;

    HEIGHT    BR_BLKS    LF_BLKS    BR_ROWS    LF_ROWS
---------- ---------- ---------- ---------- ----------
         2          1         18         17       3650
```

Tablica u primjeru ima dva stupca datumskog tipa, D1 i D2, inicijalizirana sa identičnim skupom podataka. U primjeru, simulirana su dva scenarija kreiranja indeksa. Prvi indeks T_D1 kreiran je sa PCTFREE 10%, budući da je pretpostavljeno da će buduće DML aktivnosti nad postojećim blokovima indeksa biti manjeg opsega, pa je vjerojatnost blok splitova relativno niska.

Drugi indeks T_D2 kreiran je za scenario u kojem se očekuje povećani unos redaka u postojeće blokove indeksa, te je u svakom leaf bloku rezervirano PCTFREE 50% prostora za budući unos podataka, kako bi se minimizirao broj blok split operacija. U ovakvom scenariju, ukoliko se novi unos podataka bude vršio samo unašanjem novih vrijednosti na desnoj strani indeksa, slobodan prostor u postojećim blokovima, neće nikada biti iskorišten. Primjer ukazuje na opasnost da se zbog pogrešne pretpostavke udvostruči zauzeće diskovnog prostora, smanji efikasnost iskorištenosti DB buffer cachea, te potencijalno poveća broj logičkih i fizičkih I/O operacija.

Primjer 5.: Utjecaj razine konkurentnosti (broja istovremenih transakcija nad istim indeks blokom) na raspoloživost prostora za pohranu indeks zapisa u blokovima indeksa.

U ovom pomalo ekstremnom primjeru, biti će prikazan negativan utjecaj povećane konkurentnosti DML operacija na raspoloživi prostor unutar indeks blokova. U primjeru će biti simuliran istovremeni insert novih redaka iz 200 istovremenih sesija. Budući će sesije koristiti monotono rastuće vrijednosti iz oracle sequenca, svi indeks zapisi će morati biti pohranjeni u isti indeks blok, što će dovesti do ekspanzije ITL slotova sve do maksimalne vrijednosti dozvoljene unutar pojedinog leaf bloka. Cilj testa je ispitati što se događa sa ITL slotovima nakon što se blok u kojem je nastala prvobitna ekspanzija potpuno ispuni te bude raspodijeljen (block split). Kako bi pokrenuli istovremeni upis iz 200 različitih sesija, koristimo database scheduler (DBMS_JOB paket) za istovremeno pokretanje sesija koje će izvršiti insert, te zatim ostati blokirane neposredno prije potvrde transakcije (commit) na semaforu koji

će biti implementiran kao exclusive lock putem DBMS_LOCK PL/SQL paketa. Prvi korak je kreiranje sequenca, tablice i indeksa:

```
create sequence hroug_seq cache 500;

create table t
(
 n1 number,
 vc varchar2(10)
);

create index t_n1 on t(n1);
```

Glavni semafor implementiran je kao exclusive lock putem DBMS_LOCK paketa:

```
declare
 l_lhandle number;
begin
 l_lhandle :=
 dbms_lock.request(
  id=>0,
  lockmode=>dbms_lock.x_mode,
  release_on_commit=>true);
end;
```

PL/SQL procedura koju će istovremeno pokrenuti 200 sesija, nakon izvršenog inserta, a prije potvrde transakcije (commit) ostaje blokirana na DBMS_LOCK semaforu:

```
create or replace procedure p_insert(p_jobid in number)
is
 l_lhandle number;
begin
 dbms_application_info.set_client_info('jobid: '||p_jobid);
 /* CJQ ce pokrenuti Jxxx worker procese, koji
  * ce zatim jedan po jedan insertirati redak
  * i cekati na semaforu dok parent ne commitira X lock,
  * sto bi trebalo bumpati ITL slotove
  */


 insert into t (n1, vc) values (hroug_seq.nextval, 'job'||p_jobid);

 l_lhandle :=
 dbms_lock.request(
  id=>0,
  lockmode=>dbms_lock.s_mode,
  release_on_commit=>true);

  commit;

  dbms_application_info.set_client_info('');
end;
```

Uz pomoć sljedećeg PL/SQL koda, izvršeno je istovremeno pokretanje 200 sesija putem DBMS_JOB paketa:

```
declare
 l_job number;
begin
 for x in 1..200
 loop
  dbms_job.submit(l_job, 'p_insert('||x||');');
 end loop;
 commit;
end;
```

Nakon pokretanja, očekivano, neke od sesija ostale su blokirane čekajući na slobodan ITL slot u trenutku kada ostale sesije alociraju svih 169 slotova (hard limit ITL slotova za 8 KB veličinu bloka):

```
EVENT                                                            COUNT(*)
---------------------------------------------------------------- ----------
...
enq: TX - allocate ITL entry                                             32
...
```

Po završetku masovnog istovremenog inserta, indeks se još uvijek sastoji od samo jednog leaf bloka, u kojemu je alocirano 169 ITL slotova, koji zauzimaju vrlo velikih 169 x 24 = 4056 byetova (cca 50% veličine bloka). Ipak, u bloku još uvijek ima 1089 byteova slobodnog prostora, kojega je potrebno popuniti kako bi mogli izazvati diobu bloka i ispitati što će se dogoditi nakon diobe bloka sa ITL slotovima.

Nakon što sve sesije završe insert, indeks treedump potvrđuje da je insertirano ukupno 200 redaka:

```
----- begin tree dump
leaf: 0x1005141 16798017 (0: nrow: 200 rrow: 200)
----- end tree dump
```

dok je na simboličkom pregledu bloka vidljivo da je alocirano ukupno 169 ITL elemenata:

```
Block header dump:  0x01005141
 Object id on Block? Y
 seg/obj: 0x11f9b  csc: 0x00.1fa843  itc: 169  flg: -  typ: 2 - INDEX
     fsl: 0  fnx: 0x0 ver: 0x01

 Itl           Xid                  Uba         Flag  Lck        Scn/Fsc
0x01   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x02   0x00d3.001.00000002  0x01403a52.0000.03  C---    0  scn 0x0000.001fa687
0x03   0x00d9.001.00000002  0x01403ab2.0000.03  C---    0  scn 0x0000.001fa6c2
0x04   0x0026.002.000000f0  0x01400d2e.0041.23  C---    0  scn 0x0000.001fa572
0x05   0x0023.000.000000f2  0x01400368.0058.36  C---    0  scn 0x0000.001fa578
...
...
0xa6   0x00c5.001.00000002  0x01401672.0000.03  C---    0  scn 0x0000.001fa542
0xa7   0x00c6.001.00000002  0x01401882.0000.03  C---    0  scn 0x0000.001fa552
0xa8   0x00c7.001.00000002  0x01401892.0000.03  C---    0  scn 0x0000.001fa557
0xa9   0x00c8.001.00000002  0x014018a2.0000.03  C---    0  scn 0x0000.001fa559
Leaf block dump
===============
header address 47928268548612=0x2b972b6bea04
kdxcolev 0
KDXCOLEV Flags = - - -
kdxcolok 0
kdxcoopc 0x80: opcode=0: iot flags=--- is converted=Y
kdxconco 2
kdxcosdc 0
kdxconro 200
kdxcofbo 436=0x1b4
kdxcofeo 1525=0x5f5
kdxcoavs 1089
kdxlespl 0
kdxlende 0
kdxlenxt 0=0x0
kdxleprv 0=0x0
kdxledsz 0
kdxlebksz 4024
row#0[4012] flag: ------, lock: 0, len=12
col 0; len 2; (2):  c1 02
col 1; len 6; (6):  01 00 51 39 00 00
...
```

Nakon inserta novih stotinu redaka, treedump pokazuje da sada postoje 2 leaf bloka, kao rezultat alokacije prostora kroz blok split operaciju:

```
insert into t
select rownum+200, null
from dual
connect by level <= 100; commit;

----- begin tree dump
branch: 0x1005141 16798017 (0: nrow: 2, level: 1)
   leaf: 0x1005142 16798018 (-1: nrow: 272 rrow: 272)
   leaf: 0x1005143 16798019 (0: nrow: 28 rrow: 28)
----- end tree dump
```

Simbolički dump otkriva da je svih 169 ITL slotova klonirano kroz blok split u novi blok. Alokacija većeg broja ITL elemenata predstavlja kompromis između veće konkurentnosti DML naredbi, nauštrb slobodnog prostora u blokovima indeksa. Dovoljan je samo jedan ekstremni događaj, pa da nepoželjan efekt kroz blok split operacije nastavi propagirati i u ostale blokove indeksa, kao što je vidljivo iz simboličkog prikaza drugog po redu leaf bloka dodijeljenog indeksu:

```
Block header dump:  0x01005143
 Object id on Block? Y
 seg/obj: 0x11f9b  csc: 0x00.1fab76  itc: 169  flg: O  typ: 2 - INDEX
     fsl: 0  fnx: 0x0 ver: 0x01

 Itl           Xid                  Uba         Flag  Lck        Scn/Fsc
0x01   0x00ba.005.00000002  0x01401344.0000.02  CB--    0  scn 0x0000.001fab76
..

Leaf block dump
===============
header address 47056972241412=0x2acc4e1d5a04
kdxcolev 0
KDXCOLEV Flags = - - -
kdxcolok 0
kdxcoopc 0x80: opcode=0: iot flags=--- is converted=Y
kdxconco 2
kdxcosdc 1
kdxconro 28
kdxcofbo 92=0x5c
kdxcofeo 3661=0xe4d
kdxcoavs 3569
kdxlespl 0
kdxlende 0
kdxlenxt 0=0x0
kdxleprv 16798018=0x1005142
kdxledsz 0
kdxlebksz 4024
```

ZAKLJUČAK

Ažuriranjem podataka u tablicama automatizmom se mijenjaju i podaci pohranjeni u pripadajućim indeksima. Specifični obrasci izmjene podataka mogu dovesti do povećane fragmentacije prostora unutar blokova indeksa. Detaljnije razumijevanje internih struktura indeksa i metoda kojima dobivamo uvid u te strukture pomaže nam u prepoznavanju potencijalnih problema i donošenju odluka o potrebnim korektivnim akcijama temeljenih na dokazima i činjenicama umjesto, u pravilu, pogrešnih pretpostavki.