ICOD-2

PROCEEDINGS

Second
International Conference
On
Databases

Churchill College, Cambridge
August 30 - September 3 1983

Edited by
S. M. Deen
University of Aberdeen
and
P. Hammersley
The Computer Journal

Published by
WILEY HEYDEN LTD
on behalf of
THE BRITISH COMPUTER SOCIETY
PERFORMANCE COMPARISON BETWEEN B*-TREE AND PREFIX BINARY TREE INDEX ORGANIZATIONS

Nguyen Hoan Phuco*, M. Becker**, P. Sevray*

* CII - Honeywell Bull - 94, avenue Gambetta - 75020 PARIS
** ERA 592 - Institut de Programmation - Universite PARIS VI, 4, place Jussieu - 75230 PARIS CEDEX 05

B*-tree is a well-known technique for file index structuring. It is implemented in a very large range of computers by almost all the manufacturers. Prefix binary tree is a new method for index organization. It is used in IBM system 38 series computers. However, only its "logic structure" is known. In this paper we compare, both theoretically and experimentally, B*-tree and prefix binary tree index performance in a virtual memory environment, with paging technique.

The theoretical performances of the 2 trees are evaluated in order to give some gross guidelines on their real performances, especially on the one of the prefix binary tree index organization. The "best configurations" of the "index page trees" are compared, under the hypothesis of completely uniform distribution of keys.

Then, we propose a paging algorithm for prefix binary tree index, which maybe used in the general case, without any hypothesis on key distribution. Experiences have been performed with B*-tree and prefix binary tree indexes built on the same set of files. Experimental results confirm in a certain degree, the preceding theoretical analysis.

1. INTRODUCTION

Most of the index organizations implemented on computers use a B*-tree structure. A good introduction to this structure can be found in [3] and [5]. The performance of B-tree has been studied in [7] and in several other papers ([1],[11],...).

Prefix binary tree (P. B. tree) is a new index organization developed in the IBM System 38 series computers [4]. It is a particular case of the prefix B-tree [2] (do not confound prefix B-tree and prefix binary tree) and is characterized by a total compression of keys: each character of a given key is stored once and only once in the index.
However, only the "logic structure" (i.e. structure without any paging consideration) of the IBM System 36 index is known [4].

In this paper, we are mostly interested in the proposition of a paging algorithm for P.B.tree index and in the performance comparison between a B*-tree index with its standard paging technique and P.B.tree index paged by our algorithm.

The theoretical performance comparison is performed through the "best configurations" of B*-tree and P.B.tree indexes. We define a best configuration as the one obtained with the best paging method under the hypothesis of uniform key distribution. The best paging method that may or may not exist, provides a minimal index level number and a minimal volume for each of the highest (1) levels. Best configuration can be used as a reference configuration for a given index organization in theoretical performance comparison. This second utilization of best configurations gives us some hints about expected advantages or disadvantages of P.B.tree against B*-tree.

Our proposed algorithm for allocating P.B.tree index pages can be applied in all cases without any hypothesis on the key distribution. Experiences performed with B*-tree indexes (paged by a standard method) and P.B.tree indexes (paged by our algorithm) confirm, in some aspects, our preceding best configuration analysis and show the P.B.tree index is more performant in storage requirement for index highest levels and in direct access cost, while B*-tree is more efficient in sequential access by keys.

In the whole paper, the CPU time cost in index treatments is measured through the number of required I/O operations, which is its major factor. Therefore, costs of searches within a page are not considered.

2. PREFIX BINARY TREE

We call the "logic structure of a tree" or "logic tree", the structure where each node consists of one index entry, and, the "page structure of a tree" or "page tree" the overall structure formed by index pages.

IBM first introduced, in their System 36 series computers, a new method for index organization (that we call prefix binary tree or P.B.tree [8] [9]) but did not give any details about index paging method nor index entry format [4].

We try to describe, in this paragraph, the logic structure of the P.B. tree and to define a format for each P.B.tree index entry.

2.1 P.B.tree logic structure

The logic structure of a prefix binary tree (fig.1) is completely determined when the values of the keys and the character code (EBCDIC, ASCII, ...) are known. The logic tree is a binary tree such that each node usually contains only a part of the key (and not necessarily the whole key). This part is a prefix of keys of all sons of the node. It is common to all these keys.
The decision to go to the right or to the left son node, on searching, 
depends only on the test of one bit of the key. The part of the key 
which is contained in the node is not duplicated, i.e., the son node 
will contain the part of the key which is different from the part 
which is stored in the father node. So the part of the key may be 
empty and the entry will have a variable length. A node contains the 
"number of the bit which has to be tested". This number, between 0 
and 7, is the rank of the first bit from the left, which is not common to 
the two sons of the node. For example, if an index contains 2 keys AGE 
and AGA, the part of the key which is contained in the father node is 
AG and the number of the bit which has to be tested is 5, if the code 
is EBCDIC, since for this code A = 11000001 and E = 11000101.

(i) In the following, we shall say that a level is higher if it is 
nearer the root.

If 2 keys are AGE and AGES, the father node key is AGE and the number 
of the bit which has to be tested is the first bit from the left which 
is not common to "blank" and "5" (if the code is EBCDIC, it is bit 
number 0). In fig.1, we give for each node: the part of the key, the 
number of the bit which has to be tested, the left and right son 
pointers and the father pointer.

![Tree Diagram]

Fig. 1: A prefix binary tree index with 4 keys: NOM, AGE25, NO-8S, 
AGE*.

The searching of a given key always begins from the root of the tree. 
For each node, it is necessary to compare the part of the key which is 
contained in the node against the corresponding part of the key which 
is searched. If those parts are different, the searched key does not
exist in the index, if they are the same, it is necessary to test the
bit whose rank is the "number of the bit which has to be tested" and
which belongs to the byte of the key following the part which has
already been compared. If this bit is 1, the search has to be
continued on the right branch and if it is 0, the search has to be
continued on the left branch. The right son is indicated by the right
pointer, and the left pointer indicates the left son.

Finally, let us note that prefix binary tree is a dense (or secondary)
index structure (see [5] for definition of dense index), while B*-tree
may be a dense or non-dense index structure.

2.2 Proposal of a format for P.B.tree index entry

We assume that each entry contains the following fields:

(1) Leaf indicator: 1 bit tells if the node is a leaf or not.

(2) Key indicator: 1 bit tells if a part of the key exists in the
    node or not.

(3) Left indicator: 1 bit tells if the left son is inside the page or
    outside the page.

(4) Right indicator: 1 bit tells if the right son is inside the page.

(5) Number of the bit which has to be tested: we already explained
    this (3 bits).

(6) Length of the part of the key: 1 byte.

(7) Part of the key: (between 1 and 250 bytes).

(8) Left son pointer and right son pointer: (either 2 or 3 bytes).
    When the son is inside the page, the pointer contains his
    address. The page is limited to 64K. The length of the pointer is
    2 bytes. When the son is outside the page, the pointer contains
    its page number. The length of the pointer is 3 bytes. The page
    number is assumed to be between 0 and 2^{24} - 1. In this case the
    son is the root of the subtree contained in the page.

(9) Data pointer: 4 bytes give the page number and the record number
    inside the page.

(10) Father pointer: If the node is the root of the subtree contained
    in the page (only one subtree per page), the father pointer
    contains the full address (5 bytes) of the father: 3 byte page
    number and 2 byte address inside the page. But generally the
    father belongs to the same page as the son and the pointer only
    contains the 2 bytes of the father address inside the page. So,
    we shall consider that the mean length of this pointer is 2
    bytes.
When the leaf indicator tells that the entry is a leaf, there is no further bit test and no son, so, the fields 3, 4, 5 and 6 do not exist. If the entry is not a leaf, field 9 does not exist. When the key indicator shows that there is no part of the key, the fields 5 and 6 do not exist.

3. THEORETICAL PERFORMANCE COMPARISON BETWEEN B*-TREE AND P.B.TREE

Real page structures of B*-tree and P.B.tree are heavily dependent on key distribution, key insertion order and paging method (for P.B.tree). So theoretical structure prediction is very hard in the general case, especially for P.B.tree. We shall compare the theoretical performances of the 2 trees through their "best configurations".

We define the best configuration of an index as the one obtained under the hypothesis of uniform key distribution and paged by one of the best paging algorithm, i.e. an algorithm which minimizes the number of I/O operations on repeated searching. This algorithm will minimize the number of index levels, and for a given number of levels, it will try to store in main memory as many levels as possible. The highest levels are stored in main memory. So it will try to minimize the storage space of the highest levels. Each lower level will be filled with a maximum of index entries in order to alleviate higher level size. The hypothesis of equidistributed keys is not excessive. Indeed, parameters obtained from a perfectly balanced tree are considered to be a good approximation of the ones obtained from a general tree. This fact was verified experimentally and is especially true for large files. Besides, uniform key distribution is assumed by several authors in their theoretical analysis ([11], [2], [10], ...).

Let us note that even when the keys are not equidistributed, the best paging algorithm provides a balanced page tree.

3.1 Best configuration of a P.B.tree index

In the following, we shall calculate the values of parameters characterizing the index structure when the key values are random and equidistributed, so we assume that the logic tree is perfectly balanced.

3.1.1 Calculation of the logic tree parameters

The calculated parameters of the logic tree are the height of the tree and mean length of an index entry. Input parameters are:

N: the total number of records in the file, and
l: the length of a key (or the mean length if key lengths are variable).

Since the tree is assumed to be balanced the leaves belong to the last level.
Let R be the height of the logic tree, i.e. the number of nodes it is necessary to go through, from the root to reach a leaf:
\[ 2^{R-1} = N \]
or: \[ R = 1 + \log_2 N \]

We assume that the root of the tree has no part of the key, otherwise every node would have the same prefix and the keys would not be equidistributed. Besides, the uniform key distribution hypothesis also allows us to assume that the 2 sons (if they exist) of any index entry are both inside or outside the page which contains the father entry.

The calculations will be different for a high level and a low level page. We shall say that a page is a low level page when it contains leaves, otherwise it is of high level.

Let l be the length of the key. We shall consider 2 cases:

A) \( l \leq R - 1 \): when searching we approximately go through l nodes with a part of the key (1 byte) and \((R-l)\) nodes without any part of the key. So the probability for a node to contain a part of the key is \(1/R\).

A.1) Page of high level (i.e. without any leaf)

A.1.1) If the sons are inside the page the length of an entry is:

- indicators: 1 byte.
- 2 sons and the father pointers: 3x2 bytes.
- length of the part of the key: 1 byte.
- part of the key: 1 byte.

So the mean length of the entry, when conditioning by the event: "the sons are inside the page" is: \(7 + 2l/R\) bytes.

A.1.2) If the sons are outside the page the 2 son pointers have 3 bytes instead of 2 bytes, so the mean value of an entry length, when conditioning by the event: "the sons are outside the page" is: \(9 + 2l/R\) bytes.

The tree is binary, thus if a page includes \(n\) entries, \((n+1)/2\) entries will have outside sons, and the probability for an entry to have an outside son will be approximately \(1/2\). The mean length \(r\) of an high level entry is:
\[ r = (7+2l/R)/2 + (9+2l/R)/2 = 8 + 2l/R \] bytes

A.2) Page of low level (containing leaves)

The similar calculation gives: the mean value of an entry, when conditioning by the event: "the sons are inside the page" is equal to the mean value of an entry, when conditioning by the event: "the sons are outside the page", it is: \(7 + 2l/R\) bytes. So the expected length of a low level entry is \(r'\):
\[ r' = 7 + 2l/R \] bytes.

B) \( l > R - 1 \): On the searching path, each node but the root has a part of key which is \( \lceil l/(R-1) \rceil \) or \( \lceil l/(R-1) \rceil \) bytes long. We respectively denotes by \( \lceil x \rceil \) and \( \lfloor x \rfloor \) the largest integer value which is not more than \( x \), and the smallest integer value which is not less than \( x \). Let \( t \) be the rest of the division of \( l \) by \( R-1 \). We can say that \( t \) nodes have a part of the key which is \( \lceil 1/(R-1) \rceil \) long, and \( (R-1-t) \) nodes have a part of the key which is \( \lfloor 1/(R-1) \rfloor \) bytes long and one node, the root, has no part of the key. The similar calculations as the preceding ones give:

\[
\begin{align*}
\quad r &= \left( (7+(1+\lceil l/(R-1) \rceil)/2) (R-1-t)/R + (1+\lceil l/(R-1) \rceil \right) t/R \\
&+ \left( (9+(1+\lceil l/(R-1) \rceil)/2) (R-1-t)/R + (1+\lceil l/(R-1) \rceil \right) t/R \\
\quad r' &= 8 + (1+\lceil l/(R-1) \rceil) (R-1-t)/R + (1+\lceil l/(R-1) \rceil) t/R
\end{align*}
\]

and:

\[
\begin{align*}
\quad r' &= 7 + (1+\lceil l/(R-1) \rceil) (R-1-t)/R + (1+\lceil l/(R-1) \rceil) t/R
\end{align*}
\]

3.1.2 Calculation of the page tree parameters

The parameters of the page tree that will be calculated are the height of the page tree (i.e. the number of index levels) and the number of entries per page at each index level. 3 input parameters are required for the calculation:

- the index page size \( J \),
- the maximal occupancy rate \( \sigma \) allowed for a high level page, and
- the maximal occupancy rate \( \sigma' \) allowed for a lowest level page.

\( \sigma \) and \( \sigma' \) are defined in order to introduce a free space in each page and to prevent from frequent page splittings (\( \sigma' \) is usually smaller than \( \sigma \) because the probability of having an insertion is higher for the lowest level).

Expected values of the maximal numbers of entries for high level page and for a lowest level page are respectively:

\[ Z = \lceil J/r \rceil \quad \text{and} \quad Z' = \lceil J/r' \rceil \]

Let \( M \) and \( M' \) be the numbers of entries that we expect to get for one high level page and for one lowest level page respectively.

\[ M = \sigma Z \quad \text{and} \quad M' = \sigma' Z' \]

The average number of leaves contained in a lowest level page is \( (M'+1)/2 \). So the total number of lowest level pages is:

\[ N' = \lceil N/(M'+1)/2 \rceil = \lceil 2N/(M'+1) \rceil \]
with the assumption that there is one logic tree leaf for each data record, (secondary index). Let us define:

\[ L' = \frac{M+1}{2} \text{ and } L = 2L' \]

\( L' \) is the average number of leaves contained in a \( M \)-node tree and \( L \) the total number of son pages for each father page.

We now suppose the existence of a "best paging method" for the P.R. tree index which provides the following page tree structure: each index page may be occupied by \( M \) or \( M' \) entries (according to its level) with only one logic subtree per page. So, for a perfectly balanced logic tree, we can deduce that:

\[ H = \lceil \log_L M' \rceil + 1 \]

Indeed:

if \( N' = 1 \), then \( H = 1 \),

if \( 1 < N' \leq L \), it is possible to insert \( N' - 1 \) entries on the high level \( (N' - 1 \leq M) \) and there are \( N' \) pages on the lowest level, so \( H = 2 \),

if \( L < N' \leq L^2 \), 3 levels are necessary because \( N' - 1 > M \), and \( M \) is the number of entries that we expect to get; \( L^2 \) is the maximal number of lowest level pages for a 3 level tree (if \( M \) entries are inserted into each high level page); so, in this case \( H = 3 \) ....

We now try to determine the average number \( K_i \) of entries per page for each index level \( i \) \( (i = 1, \ldots, H) \).

We begin with the lowest level.

\( N/N' \) is the average number of leaves per lowest level page. So:

\[ K_H = 2(N/N') - 1 \]

Let us define:

\[ L_i = K_i + 1 \quad \forall i = 1, \ldots, H-1. \]

\( L_i \) is the number of level \((i+1)\) son pages for each level \( i \) father page. We shall determine \( K_i \) (for \( 1 \leq i \leq H-1 \)) through \( L_i \). From the definition of \( L_{H-1} \) we get:

\[ L_{H-1} = \frac{\text{number of level } H \text{ pages}}{\text{number of level } (H-1) \text{ pages}} \]

\[ L_{H-1} = N' / \lceil \log_L M' \rceil \]

Similarly,

\[ L_{H-2} = \frac{\lceil \log_L M' \rceil}{\lceil \log_L (\lceil \log_L M' \rceil / L) \rceil} \approx \frac{N' / L}{\lceil \log_L (N'/L^2) \rceil} \]

\[ \ldots \]
Finally we can write:

\[ L_i = \frac{v_{i+1}}{v_i} \quad \forall i = 1, \ldots, H-1 \]

with:

\[ v_i = \frac{N'}{l_{i+1}} \quad \forall i = 1, \ldots, H. \]

Note that \( v_1 = 1 \) and \( v_H = N' \).

3.1.3 Properties of the best P.B. tree index configuration

(1) \( \forall i = 1, \ldots, H-1, \prod_{j=1}^{i} L_j \) is an integer number and \( \prod_{i=1}^{H-1} L_i = N' \)

(2) \( \forall i = 1, \ldots, H-1, L_i \leq L. \)

(3) \( 2 \leq L_1 \leq L_2 \leq \cdots \leq L_{H-1} \leq L. \)

(4) Among all the sequences for which (1) and (2) hold, \( (L_i) \) is the smallest sequence for the relation \( \alpha \) defined by:

\[ \forall (X_i) \text{ and } \forall (Y_i), (X_i) \alpha (Y_i) \iff \forall i = 1, \ldots, H-1, \prod_{j=1}^{i} X_j \leq \prod_{j=1}^{i} Y_j \]

--- optimal index

--- any index

level 1

level 2

... 

level H

Fig. 2: Scheme of the optimal index

Property (3) proves that the page tree grows wider as the level decreases, though the maximum page utilization is limited to the chosen value. Property (4) proves that the sequence \( L_i \) is optimal (i.e. the \( L_i \) are minimal for a particular primary memory capacity). In other words, if there is any P.B. tree index paged by some another method providing an identical number of lowest level pages (property 1) and guaranteeing a minimal free space per page (property 2), then the number of pages at each index level is always greater than the one defined by our coefficients \( L_i \). (see figure 2)
3.2 Best configuration of a $B^*$-tree index

A $B^*$-tree index is an index whose page tree is structured as a $B^*$-tree with a sequential structure inside each page and a binary searching organization. A $B^*$-tree is a B-tree for which the data pointers all belong to the lowest level and each key of level $i$ is duplicated on the lower level: (level $i+1$). The lowest level pages are sequentially "chained" in order to make the sequential processing of the keys easier ( [5], [3] ).

Each node of the $B^*$-tree is a page and each index entry is formed by a (full) key and a pointer. The pointer is 3 bytes long (index page number) for high levels and 4 bytes long (data record pointer = data page number + record number) for the lowest level.

We shall use the same notations for $B^*$-tree index as for P.B.tree index but with a * sign.

So, the length of a $B^*$-tree index entry is:

\[ r^* = 1 + 3 \text{ (for a high index level page)} \]
\[ r^{*1} = 1 + 4 \text{ (for a lowest index level page)} \]

Because key insertions always happen on the lowest index level and page splitting occurs on this level more frequently than on higher levels, we can assume that the maximal occupancy rate allowed for a high level page is

\[ \sigma = 100\% \]

The following results may be easily obtained:

\[ Z^* = \lceil \sqrt{r^*} \rceil \]
\[ Z^{*1} = \lceil \sqrt{r^{*1}} \rceil \]
\[ N^* = Z^* \]
\[ N^{*1} = \lceil N/N^{*1} \rceil \]

and the height of the page tree is:

\[ H = \lceil \log N^* N^{*1} \rceil + 1 \]

Besides, the coefficients $L^*_{i}$ can be evaluated according to the following expressions, deduced from the definition of ($L^*_{i}$):

\[ L^*_{1} \ldots L^*_{H} = N \]
\[ L^*_{1} \ldots L^*_{H-1} = N' \]

\[ L^*_{1} \ldots L^*_{i} = \lceil L^*_{1} \ldots L^*_{i+1}/N' \rceil \quad \forall i = 1, \ldots H-2. \]
3.3 Theoretical results

3.3.1 General properties

If \( r' \leq r^*/2 \) and \( r \leq r^* \), then:

1. \( \forall i = 1, \ldots, H-1 \prod_{j=1}^{i} L_j \leq \prod_{j=1}^{i} L_j^* \) and \( L_1 \prod_{i=1}^{H-1} L_i = \prod_{i=1}^{H} L_i^* = N \)

2. \( H \leq H^* \)

3. \( \forall k = 1, \ldots, H \quad v_k < v_k^* \)

These properties state that as soon as \( r' \leq r^*/2 \), and \( r \leq r^* \), the best configuration of a P.B.tree index is better than the one of a B*-tree index, as far as number of page faults and storage requirements are concerned. These conditions could be nearly always fulfilled if the entry size of the P.B.tree index was reduced by removing some of the pointers (e.g., father pointer). But this might increase the processing time of each entry.

3.3.2 Numerical examples

Theoretical performances of the 2 best configurations are now compared through a particular example. Let us choose:

- \( l = 16 \) bytes (key length)
- \( J = 256 \) bytes or \( 512 \) bytes (index page size)
- \( \sigma = 0' = 100 \% \) (maximal utilization allowed for an index page)
- \( N = 500, 1000, 2000, \) or \( 5000 \) articles, (total numbers of data records)

We deliberately choose small page sizes and small numbers of articles in order to save CPU time in experiences (see Section 5). Results are given in Table 1.

In these examples we note that best configuration of a P.B.tree index always provides a number of levels which is less than or equal to the number of levels of best configuration of B*-tree index, and it always requires a smaller storage space for high index levels. The saving on space requirement for high index levels is about 55% for P.B.tree. Otherwise, the number of lowest level pages that expresses the number of I/O operations required to treat sequentially the whole file (assuming that all leaf index entries are in lowest level pages) for P.B.tree index is about 10% less than the one for B*-tree.

4. PAGING ALGORITHM FOR P.B.TREE INDEX

We now define a paging algorithm for P.B.tree index in the general case without any hypothesis on key distribution nor on index structure evolution (number of data records, key length,...)
### Table 1

Theoretical comparison between P. B. tree's and B*-tree's best configuration.

<table>
<thead>
<tr>
<th>Page Size</th>
<th>Number of Articles</th>
<th>Number of Index Levels</th>
<th>High Index Levels Size (in KB)</th>
<th>Global Index Size (in KB)</th>
<th>Nb of Pages on Lowest Index Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>256 Bytes</td>
<td>500</td>
<td>3 (3)</td>
<td>0.4 (0.9)</td>
<td>9.8 (10.8)</td>
<td>39 (42)</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>3 (3)</td>
<td>0.8 (1.7)</td>
<td>18.5 (21.7)</td>
<td>72 (84)</td>
</tr>
<tr>
<td></td>
<td>2000</td>
<td>3 (3)</td>
<td>1.5 (3.4)</td>
<td>37 (43.4)</td>
<td>143 (167)</td>
</tr>
<tr>
<td></td>
<td>5000</td>
<td>3 (4)</td>
<td>3.6 (8.6)</td>
<td>92 (108.6)</td>
<td>395 (417)</td>
</tr>
<tr>
<td>512 Bytes</td>
<td>500</td>
<td>2 (2)</td>
<td>0.2 (0.4)</td>
<td>9.4 (10.4)</td>
<td>19 (20)</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>2 (3)</td>
<td>0.4 (0.8)</td>
<td>19 (20.8)</td>
<td>37 (40)</td>
</tr>
<tr>
<td></td>
<td>2000</td>
<td>3 (3)</td>
<td>0.8 (1.6)</td>
<td>38 (41.6)</td>
<td>72 (80)</td>
</tr>
<tr>
<td></td>
<td>5000</td>
<td>3 (3)</td>
<td>1.8 (3.9)</td>
<td>94 (104)</td>
<td>176 (200)</td>
</tr>
</tbody>
</table>

Figures for P. B. tree: first lines
Figures for B*-tree: within parenthesis
High index levels = Global index - Lowest index level
Two major objectives are pursued by the algorithm: ease for sequential treatment by keys and balance for index page tree.

Input parameters are index page size (\(d\)) and maximal occupancy rate (\(d'\)) allowed for a lowest level page.

This technique is aimed at getting all the leaf index entries on the lowest level. Some features of the algorithm are:

*The index is built by successive insertions from an empty index.

*The first index page is considered as a lowest index level page. So the maximal occupancy rate \(d'\) is applied when the page is to be split. Higher index levels are not concerned by free space reservation (because our algorithm will force leaf index entries to belong to the lowest index level).

*Each data record insertion requires the creation of 2 index entries: one entry \(A\) containing a pointer to the new record and its father entry. Both 2 entries are inserted in the same page as entry \(A\)'s brother.

*Generally, if a page \(P\) has to be split, it is divided into 3 parts:
  - the root entry \(R\) of \(P\) is displaced into the page of immediately higher level. This page has to be created if it does not exist. (Remind that there is only one subtree, hence one root entry, per page).
  - the left subtree of \(R\) remains in page \(P\).
  - the right subtree of \(R\) is transferred into a new page on the same index level as \(P\).

*This page splitting technique is similar to the B*-tree's one.

*If the entry \(A\) insertion occurs in a high index level page \(P\) (a high index level is, by definition a not lowest level), then \(A\) is systematically put in a lowest level newly created page; this is true, if page \(P\) is overflowed or if it is not. If it is overflowed (even after having put \(A\) on a lowest level page), and if \(A\)'s father is also the subtree root of \(P\), then \(A\)'s father is simply transferred into a page of immediately higher level, (this page has to be created if it does not exist).

The detailed algorithm is the following:
PERFORMANCE COMPARISON BETWEEN B*-TREE AND PREFIX BINARY TREE INDEX ORGANIZATIONS

Step 1: (A) <- index entry pointing to new data record (entry to be inserted),
    (B) <- (A)'s father entry (entry to be inserted),
    (A') <- (A)'s brother entry,
    (P) <- old page containing (A'),
    (R) <- root entry of subtree contained in (P),
    (LS) <- left subtree of (R),
    (RS) <- right subtree of (R).
Insert (A) and (B) into (P),
If (P) = lowest level page, go to step 2,
Create a new page (P') on the lowest level,
Move (A) into (P').

Step 2: If (P) is not overflowed, end,

Step 3: If (P)'s father page exists and is of (P)'s immediately higher level go to step 4,
    Create a new (P*) on the immediately higher level of (P),

Step 4: Move (R) into (P*),
    If (RS) and (LS) are not in the same page, go to step 5,
    Create a new page (P*'') on the same level as (P),
    Move (RS) into (P*'').

Step 5: If (P)'s father page is not overflowed, end,
    (P) <- (P)'s father page,
    Go to step 3.

*Contrary to what is true for insertion, a data record deletion implies 2 index entry deletions: the entry pointing to the deleted data record and its father entry. Numerous deletions may cause page underoccupancy (page underflow) and page reunions may be then considered. But a reunion can be performed only on two adjacent pages on a same index level (i.e. for two adjacent brother pages).

5. EXPERIMENTAL RESULTS

Experiences have been performed to compare the performance of P.B.tree, with our paging algorithm, against that of B*-tree.

Input parameters are the same as in 3.3.2. In fact, for a given number of articles, we created 2 files, with 2 different key distributions, in order to observe the relations between key distribution and index page tree structure. For each file we built a P.B.tree and a B*-tree index. The main results are the number of index levels and the high index level size. The number of levels gives us the number of I/O operations for a direct access by key. The high level sizes give us the main memory space required to store permanently high index levels in order to reduce I/O operations in repeated searching.

Details are given on table 2 and 3.
<table>
<thead>
<tr>
<th>Level</th>
<th>Type</th>
<th>Size (Octets)</th>
<th>Key Distribution</th>
<th>Number of Levels</th>
<th>Page Number</th>
<th>Occupancy Rate of One Page</th>
<th>Storage Space of 0 First Levels (X Octets)</th>
<th>Experimental Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F.B. tree: first line results</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>B-Tree: within parentheses</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Page size: 256 octets</td>
</tr>
<tr>
<td>TABLE 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------</td>
<td>-----------------</td>
<td>-----------------</td>
<td>-----------------</td>
<td>-----------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5000</td>
<td>2000</td>
<td>1000</td>
<td>500</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>non uniform</td>
<td>uniform</td>
<td>non uniform</td>
<td>uniform</td>
<td>non uniform</td>
<td>uniform</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 (3)</td>
<td>3 (3)</td>
<td>3 (3)</td>
<td>3 (3)</td>
<td>3 (3)</td>
<td>3 (3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 (3)</td>
<td>2 (3)</td>
<td>2 (3)</td>
<td>2 (3)</td>
<td>2 (3)</td>
<td>2 (3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>116 (30%)</td>
<td>99 (30%)</td>
<td>86 (150)</td>
<td>25 (130)</td>
<td>17 (70)</td>
<td>9 (57)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>401 (39)</td>
<td>351 (30)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11 (16)</td>
<td>12 (16)</td>
<td>5 (8)</td>
<td>4 (7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3 (4)</td>
<td>2 (3)</td>
<td>45 (12)</td>
<td>41 (2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>54% (63%)</td>
<td>60% (67%)</td>
<td>60% (53%)</td>
<td>54% (51%)</td>
<td>60% (69%)</td>
<td>54% (50%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(54%)</td>
<td>(63%)</td>
<td>(60%)</td>
<td>(54%)</td>
<td>(60%)</td>
<td>(54%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4,40 (5,44)</td>
<td>3,91 (5,64)</td>
<td>1,53 (2,21)</td>
<td>1,39 (2,20)</td>
<td>0,42 (1,09)</td>
<td>0,70 (1,06)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>395 (286)</td>
<td>408 (286)</td>
<td>169 (115)</td>
<td>166 (116)</td>
<td>92 (55)</td>
<td>81 (56)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>61% (68%)</td>
<td>58% (68%)</td>
<td>59% (68%)</td>
<td>69% (67%)</td>
<td>55% (71%)</td>
<td>71% (70%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(61%)</td>
<td>(68%)</td>
<td>(59%)</td>
<td>(69%)</td>
<td>(55%)</td>
<td>(71%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>128,10 (107,32)</td>
<td>148,95 (110,38)</td>
<td>51,42 (41,34)</td>
<td>51,54 (41,34)</td>
<td>26,31 (20,63)</td>
<td>29,39 (20,63)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**P.B. tree : first line results**

**B- tree : within parenthesis**

**Page size = 512 octets**
In these experiments, uniform key distribution means that each key character is chosen randomly on the set of 36 alphanumeric characters and non uniform key distribution means that each key character is chosen amidst a subset of some alphanumeric characters (from 1 to 4 characters). Note that a real uniform key distribution must be a distribution over 256 possible particular signs and characters.

These results show that:

(1) The number of index levels with P.B.tree never exceeds the one with B*-tree and in some cases, this number is smaller for P.B.tree (e.g. It is true when page size = 256 bytes, number of articles = 2000 or when page size = 512 bytes and number of articles = 500).

(2) The fan-out is always greater for P.B.tree.
   It appears from (1) and (2) that to offer a same number of I/O operations in direct access, B*-tree requires a greater storage space for high index levels in main memory. In other words for a given opoco in main memory (dedicated to high levels), the number of I/O operations per direct access on P.B.tree index is always less than or equal to the number on B*-tree index. For example if the whole index but the lowest level is stored in main memory, P.B.tree with our paging algorithm allows a main memory space saving of about 35% to 50% with a mean value of about 40%, against B*-tree. In the theoretical analysis with best configurations (Section 3.3) this gain is about 55% and this figure must be considered as a limit. Our paging algorithm is not very far from the best one, if only direct access is concerned.

(3) The number of lowest level pages is always greater for P.B.tree.
   In our examples, P.B.tree requires about 30% to 70% (with mean of about 50%) more lowest level pages than the B*-tree. The best configuration analysis conversely gave an advantage of about 10% for P.B.tree in the number of lowest level pages. This difference is due to key distribution influence on P.B.tree structure and to our paging algorithm that systematically creates lowest level pages for leaf entries. This result means that our paging algorithm needs more improvements on sequential treatments.

(4) Two other advantages of B*-tree against P.B.tree can be observed in our examples:
   -B*-tree is more efficient in page utilization: the occupancy rate of an index page is about 68% for B*-tree and about 58% for P.B.tree. (Note that the page occupancy rate for B*-tree is well known: [11].
   -B*-tree index size is smaller than P.B.tree index size. But, the whole index is rarely stored in main memory, because of its size. So, in general, only the high index levels are permanently kept in main memory and in this case, P.B.tree is more efficient (see (1) and (2)).
(5) Concerning key distribution, we know that $B^*$-tree is "robust" against a wide range of key distribution. This fact is confirmed by our experiments. Conversely, as expected, the P.B.tree is more affected by variation of key distribution. Some remarks emerge from numerical results:

- P.B.tree index with uniform key distribution is always more "fanned out" than with non uniform key distribution, i.e. the index high levels size is smaller.

- But the total index size is greater for uniform key distribution. This result comes from the structure of P.B.tree: key prefixes are stored once and only once in the index and uniform key distribution possesses more different prefixes than non uniform distribution.

6. CONCLUSION

It is well known that the $B^*$-tree structure is one of the best and most heavily used type of index. We have shown, however, that the P.B.tree may give even better performance, as far as direct access is concerned. $B^*$-tree remains an efficient index organization for sequential treatments.

Performance comparison begins with the determination for each tree of a theoretical "best configuration" obtained under the hypothesis of completely uniform key distribution. A comparison between 2 best configurations shows that for direct access requiring one I/O operation on index P.B.tree offers a smaller high index level size with about 55% less main memory storage space than $B^*$-tree. It also shows that for a sequential treatment on all index keys P.B.tree requires about 15% less I/O operations than $B^*$-tree.

Best configurations may also be used as reference to measure performance of other paging algorithm on a given index organization.

Along with the definition of a paging algorithm for P.B.tree index, an implementation of the two trees shows that the gain of P.B.tree against $B^*$-tree on main memory storage space for high index levels is about 40%, depending partly on key distribution. But, for a sequential treatment of the whole index P.B.tree requires about 50% more I/O operations than $B^*$-tree. This means that P.B.tree, with our paging algorithm is a valuable index organization as far as only direct access is concerned, but, for sequential treatments, our paging algorithm needs some improvements.
REFERENCES

[9] Nguyen Hoan Phuc, "Proposition d'algorithmes de pagination pour l'index de fichier organise en arbre binaire a prefixe et evaluation de leurs performances", University Paris VI, thesis