OVERLAPPING B+TREES FOR TEMPORAL DATA

YANNIS MANOLOPOULOS AND GEORGE KAPETANAKIS

ARISTOTELIAN UNIVERSITY OF THESSALONIKI. GREECE.

ABSTRACT

Overlapping trees keep past data unchanged and copy some subtrees to represent the new data due to insertions, deletions or updates. In this report a structure based on B+trees is proposed and some involved procedures are given. Analysis estimates the extra space needed each time some records are updated. Experimentation gives almost identical results to those produced by the analysis. Overlapping structures may be used in some applications such as for backup and recovery, for storing variations of a standard file or text files and especially for temporal databases.

1. INTRODUCTION

Conventional databases do not store many data versions, ie. updated and deleted records are physically or at least logically removed. However, nowadays there is an increasing need for keeping many data versions and making queries on past data or performing trend analysis for decision making. These systems are called temporal databases [11].

Since temporal database systems are characterized by enormous space needs, optical disks (either WORM's or not) are necessary for such applications. It has been proposed to partition the database in two different storage media [1,9]. For example, current and past data may be stored in magnetic and optical disks respectively. Whenever a record update is performed, the current record version may be moved from the magnetic disk to the optical one and the new version occupies its former place.

Work on temporal database logical design is extensive. However, work for temporal database oriented structures is limited. The only works to our knowledge are the ones by Burton and Kollias [2,3, 6], Easton [4] and Lomet [8]. Overlapping B-trees by Burton and Kollias are structures which keep past data unchanged and copy some subtrees to represent new data due to insertions, deletions or updates (or any set of these operations). In this report a structure based on B+trees is examined. Under the new storage technology perspectives it seems that B+trees as an overlapping structure may be used efficiently for future applications such as for backup and recovery, for storing variations of standard or text files and especially for temporal databases.

The structure of this work is the following. In the next section the overlapping B+tree structure is defined and some procedures, important for the structure maintenance are given. In Section 3 an estimate of the extra space needed for updating a set of records is derived analytically. In Section 4 some experimental results are reported and discussed. Finally, the conclusion follows and future extensions are mentioned.

TH0326-9/90/0000/0491/$01.00 © 1990 IEEE

2. THE STRUCTURE

B+trees of order *m* have the following properties [10]:

(a) The root node has at least 2 children and at most *m* children (unless it is a leaf).

(b) Internal nodes have at least $\lceil m/2 \rceil$ children and at most *m* children. Internal nodes contain only keys and addresses of nodes of the next lower level.

(c) An internal node with *k* children contains *k*-1 keys.

(d) All leaf nodes lie at the same level.

If the B+tree is used as a main file then leaf nodes contain the data records. The data node capacity in records is at most *DCap* and at least *DCap*/2. Otherwise, ie. if the B+tree is used for secondary indexing then leaf nodes contain pairs of keys and pointers to the main file (and apparently their capacity is the same to that of the index nodes). Therefore at the last level all the keys appear as part either of a data record or of a key-pointer pair. Leaf nodes may also contain the address of the next leaf node for sequential processing.

Here, we propose a variation of B+ trees used as a main file. We assume that individual versions of a file are represent as B+trees and we consider the overlapping B+tree structure for representing a collection of files with similar content by using common subtrees. An extra field is used to determine if a substructure is shared, *the reference count*. All nodes with a reference count greater than 1, together with all descendants of such nodes constitute shared information. In the following the structure is defined in Pascal language and the most important procedures are given. Procedure *setpointer* manipulates the reference count field, while procedures *copy* and *newnode* are used to duplicate the path of nodes from the root to the leaves due to an insertion, deletion or an update. (nn is the data node capacity, mm is the index node capacity.) The rest of the procedures may be found in the appendix [5].

```
const n=3; nn=6; m=30; mm=60;
type ref=^page;
  page=record
    refcount:integer;
    case data:boolean of
      false: (c:0..mm;
        key:array[1..mm] of integer;
        p:array[0..mm] of ref);
      true : (d:0..nn;
        cle:array[1..nn] of integer;
        info:array[1..nn] of integer);
  end;

procedure setpointer(var r:ref; q:ref);
var i:integer;
begin
if (r=nil) and (q<>nil) then
  begin
    r:=q;
    q^.refcount:=q^.refcount+1
  end
else if (r<>nil) and (q=nil) then
  begin
    r^.refcount:=r^.refcount-1;
    if r^.refcount=0 then
      begin
        if not r^.data then
          for i:=0 to r^.c do
            setpointer(r^.p[i],nil);
          dispose(r)
      end;
    r:=q
  end
else if (r<>nil) and (q<>nil) then
  begin
    setpointer(r,nil);
    setpointer(r,q)
  end
end;
```

```
procedure newnode(var r:ref; q:ref);
var i:integer;
begin
if (r<>nil) then setpointer(r,nil);
new(r);
r^.refcount:=1; r^.data:=q^.data;
if not q^.data then
  begin
    r^.c:=q^.c; r^.key:=q^.key;
    for i:=0 to mm do r^.p[i]:=nil
  end
else
  begin
    r^.d:=q^.d; r^.cle:=q^.cle;
    r^.info:=q^.info
  end
end;

procedure copy (var r:ref);
var i:integer; q:ref;
begin
q:=nil;
if r<>nil then if r^.refcount>1 then
  begin
    newnode(q,r);
    if not r^.data then
      begin
        for i:=0 to r^.c do
          setpointer(q^.p[i],r^.p[i]);
        setpointer(r,q);
        setpointer(q,nil)
      end
  end
end;
```

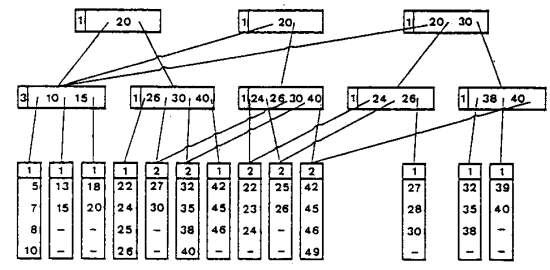

Fig.1. Overlapping B+tree of 3 versions.

Figure 1 shows an overlapping B+tree of order m=5 and data node capacity $DCap$=4. The tree has three versions; each version has two index levels and one data level. The first version contains 21 records at the last level and initially all the reference count fields are set equal to 1. The second version is produced from the first one by inserting a set of two records with key values 23 and 49 (This set is symbolized by b.) The third version is produced by inserting another two records with key values 28 and 39 in the second version (b=2 again). The reference count fields of Figure 1 depict this final state of the tree. Thus the value of this field in all root nodes is 1 and in other nodes varies from 1 to 3.

3. ANALYSIS

Assume that the block size is BS bytes and is common for index and data nodes, KS bytes is the key size, PS bytes is the pointer size and RS bytes is the record size. Assume also that the tree consists of I index levels plus one data level (the root lies in the first level). The following table summarizes the problem variables.

BS	Block Size in bytes
KS	Key Size in bytes
PS	Pointer Size in bytes
RS	Record Size in bytes
$ICap$	Index Capacity (or m)
$DCap$	Data Capacity
N	Total Number of Records
b	Number of Updates
I	Number of Index Levels

Table I. List of variables.

The index node capacity in key-pointer pairs, $ICap$ (or equivalently the variable m of the definition), is equal to:

$$ICap = \left\lfloor (BS-PS) \;/\; (KS+PS) \right\rfloor$$

while the data node capacity in records, *DCap*, is:

$$DCap = \left\lfloor BS \;/\; RS \right\rfloor$$

B+tree nodes are *ln2* full on the average [12]. Therefore, the mean value of records per data node is:

$$ln2 * DCap.$$

With a similar reasoning, the mean value of children per any index node, the fan-out, is:

$$ln2 * ICap$$

Therefore, the number of nodes at the $(k+1)$-th level, is:

$$a_{k+1} = N \;/\; (ln2 * DCap)$$

where N is the number of records.
The number of nodes at the k-th level is:

$$a_k = a_{k+1} \;/\; (\, ln2 * ICap \,)$$

and generally the number of nodes at the i-th level is:

$$a_i = a_{k+1} \;/\; (\, ln2 * ICap \,)^{k+1-i}$$

where $1 \le i \le k+1$.

Improving time and space performance via batch operations in tree structured organizations has been examined in [7]. Here, in the same respect, we assume that during a time period a set of b, out of N, distinct record updates has arrived. We are concerned only with updates, not insertions or deletions. The mean value of data nodes in which the b records belong is [13]:

$$a_{l+1} * \left(1-(1-1/a_{l+1})^b \right)$$

With a similar reasoning, expectedly these node are indexed by the following number of the l-th level nodes:

$$a_l * \left(1-(1-1/a_l)^b \right)$$

Therefore, finally the total number of either index or data nodes which will be accessed to search for the b records expectedly is:

$$\sum_{i=1}^{l+1} a_i * \left(1-(1-1/a_i)^b \right)$$

Evidently, this number of nodes will have to be duplicated to represent the new record information. Since, for simplicity we are not concerned with record insertions or deletions, it follows that the number of records and nodes per any tree version remains constant with time. Therefore, given that the number of record updates per transaction remain constant, the number of nodes to be copied remains constant with time too; the structure augmentation is linear with time.

4. EXPERIMENTATION

From extensive experimentation the following figures are extracted. Figure 2 depicts the relation between the total number of updates and the augmentation percentage of the tree under the following conditions. The file has N=4000 records, maximum index capacity *ICap*=60, while the maximum data capacity, *DCap*, takes the two values 4 and 6, and the number of updates per transaction, b, is 10 or 100. As expected for increasing b the number of nodes to be copied decreases. It is shown, also, that for decreasing *DCap* the number of nodes to be copied increases. This is explained by considering that if the data capacity increases, then the probability to access and copy a data node increases too.

Figure 3 depicts the relation between the number of updates and the augmentation percentage of the tree under the following conditions. The file has N=1000 records and the ratio *ICap*/*DCap* is constant (=10). The values taken by *ICap* are 40, 60, 80, 100 and 120. This means that the tree varies from deep and wide to shallow and narrow. Evidently, the percentage tends asymptotically to unity with increasing number of updates, b, in one transaction. Shallower and narrower (deeper and wider) is the tree, the convergence to unity is faster (slower).

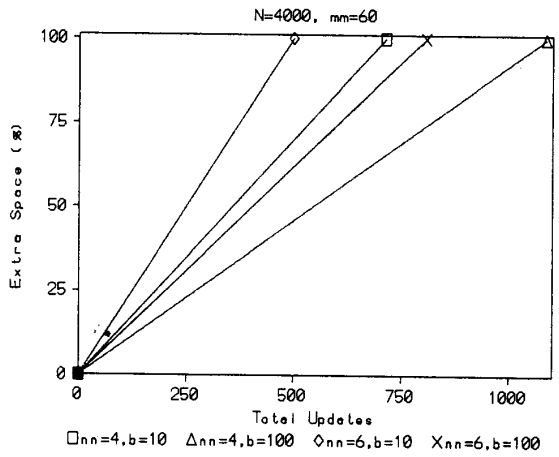

Fig.2. Extra space needed as a function of the total number of updates.

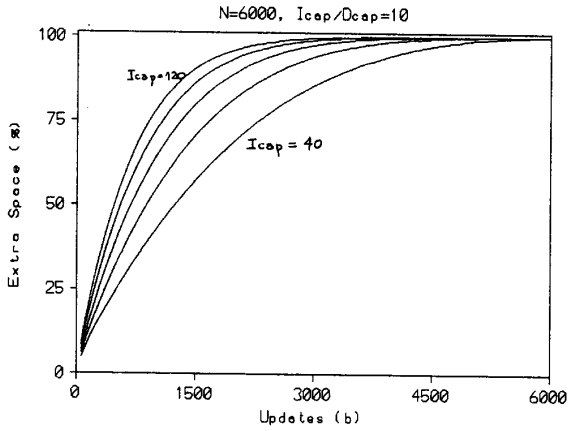

Fig.3. Extra space needed as a function of the number of updates per transaction.

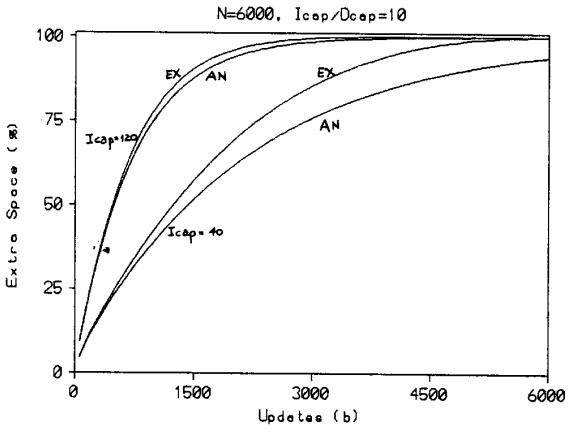

Fig.4. Experiment vs. Analysis.

Figure 4 compares experimental to analytical results. Again, the file has N=6000 records and the ratio $ICap/DCap$ is constant (=10) but $ICap$ is equal to 40 and 120 only. It is remarked that there is a small deviation which is greater for larger $ICap$'s. This is explained by the fact that the analysis is based on the expected value of the node occupancy ($ln2$) and does not consider the occupancy as a stochastic variable.

5 EPILOGUE

Database structures for applications with time support have enormous disk capacity needs. With optical disk systems it seems that in the near future temporal databases will be commercially available. In this work a data structure suitable for this environment, called overlapping B+ tree, is proposed. Some basic procedures (in Pascal language) for manipulating the structure are provided. Analysis and experimental results for the space required for successive updates of sets of records are reported. (Note, however, that the structure and the supporting algorithms may equally well be implemented in magnetic disk storage.) Future work will examine with more elaborated analysis and experimentation the structure behavior in insertions and deletions. Comparison with other variations of B+trees with time support, such as the WOBT by Easton [4] and the split tree by Lomet and Saltzberg [8], needs further research.

ACKNOWLEDGEMENT

Professor J.G. Kollias of the National Technical University of Athens had reviewed an earlier draft of this work. His untimely death is hard to believe.

REFERENCES

[1]. Ahn I. and Snodgras R.: Partitioned Storage for Temporal Databases, *Information Systems*, Vol.13, No.4, pp.369-391, 1988.

[2]. Burton F.W., Huntbach M.M and Kollias J.G.: Multiple Generation Text Files using Overlapping Tree Structures, *The Computer Journal*,Vol.28, pp.414-416, 1985.

[3]. Burton F.W., Kollias J.G., Kollias V.G and Matsakis D.G.: Implementation of Overlapping B-trees for Time and Space Efficient Representation of Collection of Similar Files, *The Computer Journal*, to appear, 1990.

[4]. Easton M.C.: Key-sequence Data Sets on Indelible Storage, *IBM Journal of Research and Development*, Vol.30, No.3, pp 230-241, 1986.

[5]. Kapetanakis G.: A B+tree Variation for Temporal Databases, *Diploma Thesis*, (in greek), EE Dept., AU of Thessaloniki, 1989.

[6]. Kollias J.G. and Matsakis D.G.: Change Area B-tree to Cover Multiple Time Periods, Proceedings of the *Eurinfo Conference*, pp. 769-774, 1988.

[7]. Lang S.D., Driscoll J.R. and Jou J.H.: Improving the Differential File Technique via Batch Operations for Tree Structured File Organizations, Proceedings of *IEEE Data Engineering Conference*, pp. 524-532, 1986.

[8]. Lomet D. and Saltzberg B.: Access Methods for Multiversion Data, Proceedings of *ACM SIGMOD Conference*, pp.315-324, 1989.

[9]. Manolopoulos Y.: Reverse Chaining for Answering Temporal Conjuctive Queries, *The Computer Journal*, submitted.

[10]. Saltzberg B.: *File Structures - An Analytic Approach*, Prentice Hall, 1988.

[11]. Snodgras R. and Ahn I.: Temporal Databases, *IEEE Computer*, Vol.19, No.9, pp.35-42, 1986.

[12]. Yao A.: Random 3-2 Trees, *Acta Informatica*, Vol.2, No.9, pp.159-179, 1978.

[13]. Yao S.B.: Approximating Block Accesses in Database Organizations, *Communications of ACM*, Vol.20, No.4, pp.260-261, 1977.

APPENDIX

```
procedure search (x,val:integer;
  var a:ref; var h:boolean; var v:item);
type item=record
       key:integer;
       p:ref; info:integer;
     end;
var r:integer; found:boolean;

procedure move1(var y,z:ref;
  t1,t2:integer; tag:boolean);
begin
if tag then
  begin
    y^.cle[t1]:=z^.cle[t2];
    y^.info[t1]:=z^.info[t2];
  end
else
  begin
    y^.key[t1]:=z^.key[t2];
    y^.p[t1]:=z^.p[t2]
  end
end;  {move1}

procedure move2(var y:ref; w:item;
  t:integer; tag:boolean);
begin
if tag then
  begin
    y^.cle[t]:=w.key; y^.info[t]:=w.info;
  end
else
  begin
    y^.key[t]:=w.key; y^.p[t]:=w.p
  end
end;
```

```
procedure bin_search (a:ref; data:
  boolean; x:integer; var r:integer;
  var found:boolean);
var l,k:integer;
begin
  with a^ do
    begin
      l:=1;
      if data then r:=d else r:=c;
      repeat
        k:=(l+r) div 2;
        if x<=key[k] then r:=k-1;
        if x>=key[k] then l:=k+1
      until r<l;
      found:=(l-r>1)
    end
end;

procedure insert(var max,half,
  num:integer);
var i:integer; b:ref;
begin
  with a^ do
    begin
      if num<max then
        begin
          num:=num+1; h:=false;
          for i:=num downto r+2 do
            move1(a,a,i,i-1,data);
          move2(a,u,r+1,data)
          end
      else
        begin   {num≥max}
          h:=true; new(b);
          b^.refcount:=1; b^.data:=data;
          if data then  b^.d:=half
          else b^.c:=half;
          if r<=half then
            begin
              for i:=1 to half do
                move1(b,a,i,i+half,data);
              if r<half then
                for i:=half+1 downto r+2 do
                  move1(a,a,i,i-1,data);
              move2(a,u,r+1,data)
            end
          else
            begin  {r>half}
              r:=r-half;
              for i:=1 to r-1 do
                move1(b,a,i,i+half+1,data);
              move2(b,u,r,data);
              for i:=r+1 to half do
                move1(b,a,i,i+half,data)
            end;
        if tag then
          begin
            v.key:=cle[half+1];
            v.info:=info[half+1];
            num:=half+1
          end
        else
          begin
            v.key:=key[half+1];
            b^.p[0]:=p[half+1];
            num:=half
          end;
        v.p:=b
      end  {num≥max}
    end    { with }
end;     { insert }

begin    { search }
copy(a);
if a=nil then
  begin
    h:=true; new(a);
    with a^ do
      begin
        refcount:=1;
        data:=true; d:=1;
        cle[1]:=x; info[1]:=val
      end;
    v.key:=x; v.p:=nil
  end
else
  begin  {a<>nil}
    with a^ do
      begin
        bin_search(a,data,x,r,found);
```

```
        if found then
          begin
            h:=false;
            if data then info[r+1]:=val
            else search(x,val,p[r],h,u)
          end
        else
          begin  {not found}
            if data then
              begin
                u.key:=x; u.info:=val;
                insert(nn,n,d)
              end
            else
              begin  {not found, not data}
                search(x,val,p[r],h,u);
                if h then insert(mm,m,c)
              end
          end
      end
  end
end;
```