P-Tree: A B-Tree Index for Lists

Ke Wang
Beng Chin Ooi
Sam Yuan Sung
School of Computing
National University of Singapore
{wangk,ooibc,ssung@comp.nus.edu.sg}

Abstract

The high frequency of applications involving large, ordered, nested lists suggests that list is the "next most" natural data type after set. A list differs from a set through positioning and nesting elements within the list. Directly supporting such position-related operations will greatly improve the performance of database systems targeting at the above applications. Unlike other attributes, the position will be changed by insertion and deletion within a list and known methods are not appropriate for indexing the position. We present an indexing structure, called the P-tree (where P for position), to index a set of lists. The P-tree generalizes the B-tree by dealing with a set of lists rather than a set of records, while preserving all the properties of the B-tree.

1. Introduction

Recently, there has been a great deal of interest in data models and access supports for "bulk" data types. Perhaps the most common such construct is the ordered, nested list (see [1, 3, 9, 11, 13] for example). The need of such lists arises from applications that store ordered relations [12], time-series data, meteorological and astronomical data streams, runs of experimental data, multi-dimensional arrays, textual information, DNA sequences, voices/sound/images/ video, etc. The ability of nesting and positioning elements within a list is the first nature of such data. A typical application would be where data can be divided into sequenced groups, and members within a group may be further subdivided into sequenced subgroups recursively. Such sequencing and nesting can go on up to an arbitrary depth. Queries and updates may be posed based on the group position, subgroup position, and so on. The position information can be specified either in the search predicate or in the search result.

For example, the X chromosome in Figure 1 (as visualized by the geneticist or biochemist) is divided into p and q parts, each part is subdivided into regions called bands, which are further subdivided, and so on, up to the resolution of 4 levels. Therefore, each of p and q parts is a nested list of depth 4. The numbers such as p11.23 along the "tube" are called locus numbers and specify the nested position information of bands on the genome. By using locus numbers, for example, the biochemist can search all the genes that lie within/overlap the region starting from 20q21.1 and ending at 20q22.3, or split the band 20q21.1 into two refined bands 20q21.1.1 and 20q21.1.2, where the number 20 is the id of the chromosome involved.

To summarize, the above types of applications require the following access supports. (a) Insertion and deletion are allowed to be performed at any position and nesting level within a list, and the position and nesting should be correctly maintained after update. (b) The primary mode of data processing is a list of oids at a time, not an oid or a set of oids at a time, as in the case of the set data. (c) The nesting depth of lists is dynamic, non-uniform, and not necessarily known in advance. A practical index structure should maintain the position and nesting of elements within a list after update, support the sequential processing as a first-class citizen operation, allow the nesting depth of lists to dynamically grow and shrink without imposing a maximum depth.

Related work. The first possible approach to indexing list data is to treat positions at each nesting level as one attribute and apply multi-dimensional indexing methods to such attributes, such as the R-tree [4], grid file [7], Buddy-tree [10], the z-ordering [8]. However, this does not work for the following reasons. First, none of these indexings addresses the change of position and nesting within a list caused by insertion or deletion. Second, these indexes assume a fixed number of attributes being indexed, thus cannot handle unbounded nesting depth. Third, they do not support inserting and deleting one list at a time, nor list-

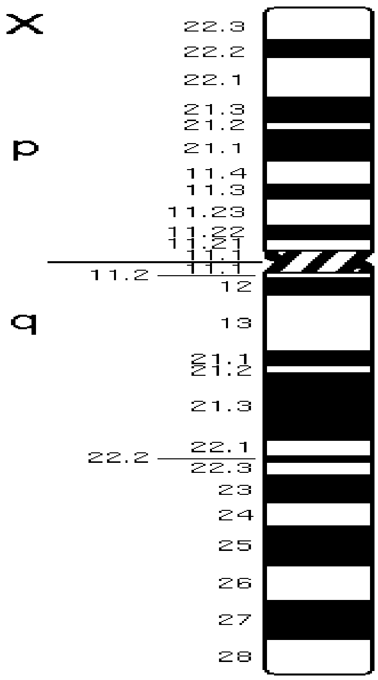

Figure 1. An ideogram of the human X chromosome, from http:// wwwgdb.gdb.org/

specific operations such as length function of lists, delete or insert a prefix or suffix, etc.

The second possible approach is applying the *ordered B-tree* for a single flat list [2, 6] to each sublist at each nesting level. The result is the repeated use of the ordered B-tree at every layer of the multi-dimensional B-tree proposed in [5], one layer for each nesting level. At the top layer, a B-tree is built on list ids, which directs the search to respective ordered B-trees for level-1 lists, based on given list ids; these ordered B-trees then direct the search, based on the given position of level 1, to respective ordered B-trees of level-2 lists, which direct the search to ordered B-trees of level-3 lists, and so on. For lists that are deeply nested and have skewed length, the whole index will become very unbalanced and the size of the database is no longer exponential in the height of the index.

Main results. We propose a B-tree like indexing structure, called the *P-tree* (P for Position), to support list operations. The P-tree allows

- to search or update lists at any position and nesting level, both directly and sequentially.
- to find the first leaf page of interest in $log_b N$ I/O page accesses, where N is the total number of elements in all lists and b is the branch factor of the P-tree.
- to restrict the change of the index to one root-to-leaf path for search and two such paths for update of an arbitrarily large portion of a list.
- nesting of lists to an arbitrary depth.

We will formally define the P-tree and its operations in the subsequent sections. Essentially, the P-tree preserves all the properties of the B-tree while providing the additional capability for handling lists. We focus on search, insert and delete operations. Other lists operations such as length function, inserting and deleting a prefix or suffix, etc. can be supported easily by the P-tree index.

2. The P-tree

2.1 The representation of lists

A *list* over some set O of oids is defined recursively as follows:

- each oid in O is a list, called a *terminal* list;
- if $l_1, ..., l_n$ are lists, $[l_1, ..., l_n]$ is a list.

A *list database* is a collection of lists $\{l_1, \ldots, l_n\}$. Each l_i in the list database is also called a *level-0* list. Each level-0 list is uniquely identified by a list id of the form #i. Each list l_i is a *child* of list $[l_1, \ldots, l_n]$, or $[l_1, \ldots, l_n]$ is the *parent* of l_i. The *position* of l_i in $[l_1, \ldots, l_n]$ is i. A *level-i* list is a child of a level-(i-1) list. The transitivity of the parent/child relationship induces the ancestor/descendant relationship in the usual way. The *nesting level* of a level-i list is i. Note that only level-0 lists are identified by ids; any descendant list of a level-0 list is identified by the position of its ancestor lists (within their respective parent lists) plus the id of the level-0 list. The *nesting depth* of a level-0 list is the maximum nesting level of all descendant lists of the list. The *length* of a list refers to the number of child lists in the list. The *size* of a list refers to the total number of oids nested at all levels in the list.

Suppose that a single letter denotes an oid. The level-0 list $l = [[klmn][[op][qr]][stuv]]$ contains three level-1 lists [klmn],[[op],[qr]],[stuv]; oids k,l,m,n are four level-2 lists; [op] and [qr] are two level-2 lists; oids o,p,q,r are four level-3 lists, and so on. The nesting depth of l is 3. The length of l is 3 because it has three child lists. The size of l is 12.

2.2. The P-tree

An entry in an internal node of the P-tree has the form $< K, Pr >$, where K is an *entry key* and Pr is the pointer to a child of the node. An entry key K of length $l(\geq 1)$ has the format of $\#i.c_1.c_2 \ldots c_l$, where #i is the id of a level-0 list in the list database and c_i (≥ 1) is the *position offset* of a level-i list, explained below. $T(Pr)$ denotes the subtree under branch Pr. Entry keys can have different length l so that nesting depth of lists can grow and shrink dynamically anywhere in a list.

Convention. In leaf nodes, we use list ids #i to mark the beginning of level-0 lists and delimiters \$i to mark the end of level-i lists. We say that some number of \$i's are *followed by* some number of \$j's if *all* these \$i's end before any of these \$j's begin. When two entry keys $K = \#i.c_1 \dots . c_l$ and $K' = \#i'.c'_1 \dots . c'_{l'}$ are tested for relationships $<, >, \leq, \geq, =, \neq$ relationships, they are treated as a $(l+1)$-tuple and a $(l'+1)$-tuple ordered by the usual lexicographic order: for two tuples $(s_1, \dots, s_p)$ and $(t_1, \dots, t_q)$, $(s_1, \dots, s_p) \leq (t_1, \dots, t_q)$ when either:

- there exists an integer j such that $s_j < t_j$ and for all $i < j$, $s_i = t_i$, or
- $p \leq q$ and $s_i = t_i$ for $1 \leq i \leq p$.

The P-tree. In the *P-tree* of degree (m, M), the following invariants hold for an internal node v with p entries $< K_1, Pr_1 >, \dots, < K_p, Pr_p >$, where $K_i = \#i.c_1^i \dots c_{l_i}^i$:

P1 In the leaf level of subtree $T(Pr_i)$, where $1 \leq i \leq p$, exactly c_1^i \$1's delimiters of list #i are followed by exactly c_2^i \$2's of list #i, which are followed by exactly c_3^i \$3's delimiters of list #i, and so on.

P2 $\#1 \leq \#2 \leq \dots \leq \#p$. (Note that entry keys K_i's are not necessarily in sorted order.)

P3 For all list ids #j in the subtree $T(Pr_i)$, if $i > 1$, $\#(i-1) \leq \#j \leq \#i$; otherwise, $\#j \leq \#i$.

P4 If v is the root, $p \geq 2$; otherwise, $m \leq p \leq M$. Usually, m is less than or equal to $M/2$.

P5 All leaf nodes are at the same level of the tree.

Intuitively, P1 implies that the nesting and position information below a node is maintained in the node; P2 and P3 imply that id's of level-0 lists are indexed; P4 ensures the utilization of storage space; and P5 ensures the balance of tree height. All leaf nodes of the P-tree are chained to facilitate forward and backward sequential scans.

Example 2.1 Consider two lists

$$l = [[abc][def][ghij]], \text{with id } \#1$$

and

$$l' = [[klmn][[op][qr]][stuv]], \text{with id } \#2,$$

where each single letter represents an oid. l contains three level-1 lists $[abc]$, $[def]$, $[ghij]$; $[abc]$ contains three level-2 lists a, b, c, which are oids; and so on. By our convention at the beginning of this subsection, l will be stored as

#1a\$2b\$2c\$2\$1d\$2e\$2f\$2\$1g\$2h\$2i\$2j\$2\$1

in the leaf level of the P-tree.

To save space, we follow another convention: the first oid in a leaf node is always ended by a delimiter; subsequently, an oid in the same leaf is ended by a delimiter only if the delimiter is different from the latest stored delimiter in the same leaf. However, this optimization is not visible to search and update operations on the P-tree. Figure 2 shows a P-tree of degree $(2, 4)$ for lists l and l'. In leaf node H, only oid a is ended by \$2 delimiter; the delimiters for b and c can be inferred from a's. In leaf node J, f is the first oid and is ended by its \$2 delimiter; g is ended by \$2 because its delimiter is different from the latest \$1.

We can verify that properties P1 through P5 are satisfied by this tree. For example, $K_1 = \#2.1.2$ in node C because in the subtree under the first branch of node C, there is exactly 1 \$1 delimiter of list #2, followed by 2 \$2 delimiters of list #2, in that order; $K_2 = \#2.0.2$ in node E because in the leaf L there is 0 \$1 delimiter (of list #2), followed by 2 \$2 delimiter, 1 for k and 1 for l; $K_1 = \#2.1.2$ in node G because in the leaf P there is 1 \$1 delimiter of list #1, followed by 2 \$2 delimiter, 1 for s and 1 for t. Note that in node A the two entry keys of list #2 have different length, i.e., 2 and 1. Similarly, in nodes C, F, G. □

3. Operators on entry keys

Before proceeding to search and update operations, we define two operators $\oplus$ and $\ominus$ on entry keys. First, let us discuss the representation of nested positions.

Position paths. The starting position of a search is specified by a *position path* of the form $\#i.s_1 \dots s_n$, where $s_i > 0$ and $n > 0$. The position path refers to the s_nth level-n list within the s_{n-1}th level-(n-1) list within the s_{n-2} level-(n-2) list, . . ., within the s_1th level-1 list of level-0 list with id #i. For example, the position path $\#i.4.6.2$ refers to the second level-3 list within the sixth level-2 list within the fourth level-1 list in the list $\#i$. We often use the entry key format of this position path. For example, the entry key format of position path $\#i.4.6.2$ is $\#i.3.5.1$, which is read as: the search (or update) starts after the first level-3 list that follows the fifth level-2 list that follows the third level-1 of list #i. $\#i.3.5.1$ is called the search key of position path $\#i.4.6.2$, whose general definition is given below.

Search keys. The *search key* of a position path $\#i.s_1 \dots s_n$ is

$$\#i.(s_1 - 1) \dots (s_{n-1} - 1).(s_n - 1).$$

To determine the current position within a list while searching or updating the P-tree, we need two operators $\oplus$ and $ominus$ on entry keys or search keys $K = \#i.c_1 \dots c_l$ and $K' = \#i.c'_1 \dots c'_{l'}$.

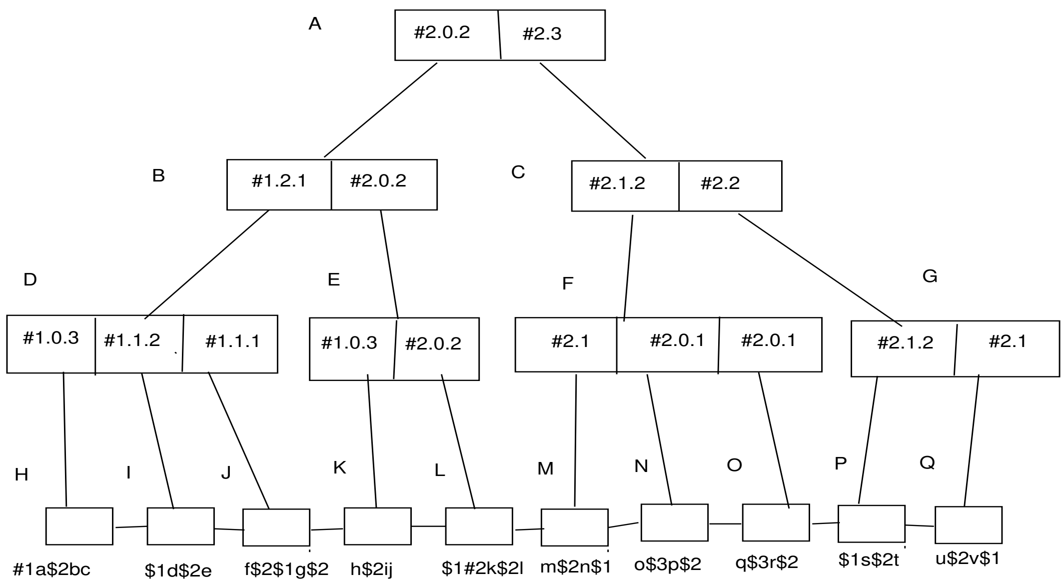

Figure 2. The P-tree structure for list [[abc] [def] [ghij]] and list [[klmn] [[op] [qr]] [stuv]]

3.1. $\oplus$ operator

If $l < l'$, append 0's at the right end of K so that K has length l'. We define

$$K \oplus K' = \#i.c_1 \ldots c_{j-1}.(c_j + c'_j).c'_{j+1}.\ldots.c'_{l'},$$

where c'_j is the left-most non-zero position offset in K'. That is, $K \oplus K'$ is the usual vector sum on position offsets but ignores all c's in K after the jth position offset. The reason for ignoring such c's is due to the semantics of an entry key stated in property P1: only the number of $i's that follow the last $(i-1) of the same list id is recorded as position offset c_i. Note that $K \oplus K'$ has the same length as K' and that $\oplus$ degenerates to ordinary + if lists are flat.

For example, $\#1.1.0.2 \oplus \#1.3.1 = \#1.4.1$, where 2 in #1.1.0.2 is ignored because $\#1.3.1$ is non-zero at nesting level 1. The justification for this computation is shown in Figure 3 (all $i delimiters refer to those of #1): if in subtree T1 1 $1 delimiter is followed by 0 $2 delimiter which is followed by 2 $3 delimiters, and if in subtree T2 3 $1 delimiters are followed by 1 $2 delimiter, then in the tree formed by left subtree T1 and right subtree T2, 4 $1 delimiters are followed by 1 $2 delimiter. In this case, the $3 delimiters are ignored due to the convention in subsection 2.2.

By using operator $\oplus$, we can accumulate position offsets (or nesting information) on the left side of a path traversed. To see this, let $K_i = \#i.c^i_1.\ldots.c^i_{l^i}$, where $1 \le i \le p$, be all entry keys in an internal node v. We define K^*_i as

$K^*_i = K_i$, if $i = 1$ or $\#(i-1) \neq \#i$,
$K^*_i = K^*_{i-1} \oplus K_i$, otherwise.

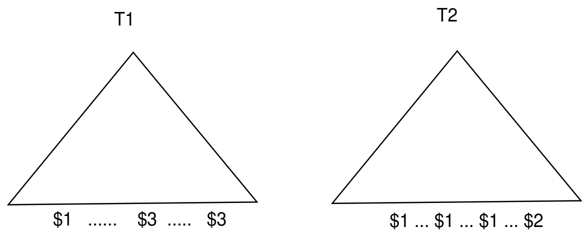

Figure 3. An illustration of $\#1.1.0.2 \oplus \#1.3.1 = \#1.4.1$

Intuitively, if $K^*_i = \#i.a_1.\ldots.a_q$, in the forest formed by subtrees $T(Pr_1), \ldots, T(Pr_i)$, there are exactly a_1 $1 delimiters of list #i, followed by exactly a_2 $2 delimiters of list #i, . . ., followed by exactly a_q $d delimiters. In particular, the right-most K^*_p gives such numbers for the whole subtree with the root v. This gives rise to the following computation of entry keys, which forms the basis for updating entry keys when performing insertion and deletion on the P-tree. Let v be an internal node and v_i be the ith child of v. Let $v.K_i$ denote the ith entry key in v. We define

- $K(v_i) = \#i.c_1 \ldots c_l$, if v_i is a leaf node in which $\#i$ is the last list id and if there are exactly c_1 $1 delimiters of list #i, followed by exactly c_2 $2 delimiters of list #i, followed by exactly c_3 $3 delimiters of list #i, . . ., followed by exactly c_l $l delimiters of list #i.

- $K(v_i) = v_i.K^*_p$, if v_i is an internal node and if K_p is the last entry key in v_i.

Example 3.1 Consider the tree in Figure 2. In node D, $K_1^* = \#1.0.3$ and the leaf node H contains 0 \$1 and exactly 3 \$2 delimiters of list #1 (2 \$2's not shown); $K_2^* = \#1.1.2$ and node I contains exactly 1 \$1 delimiter of list #1 followed by exactly 2 \$2 delimiters of list #1 (1 \$2 not shown); $K_3^* = \#1.1.1$ and node J contains exactly 1 \$1 delimiter followed by exactly 1 \$2 delimiter of list #1. In node E, $K_1^* = \#1.0.3$ and $K_2^* = \#2.0.2$. In node B, $K_1^* = \#1.2.1$, so in the subtree rooted at node D there are 2 \$1 delimiters of list #1 followed by 1 \$2 delimiters of list #1; $K_2^* = \#2.0.2$, so in the subtree rooted at node E there are 0 \$1 delimiter of list #2, followed by 2 \$2 delimiters of list #2 (1 \$2 not shown), and so on. The reader can verify that all K_i^* give correct numbers and orders of \$ delimiters in subtrees under their branches. □

3.2. The $\ominus$ operator

We now define operator $\ominus$ on entry keys or search keys $K = \#i.c_1 \ldots c_l$ and $K' = \#i.c'_1 \ldots c'_{l'}$. If $l' < l$, append 0's to the right end of K' so that K' has length l. Let $c_j - c'_j$ be the first negative difference from the left; $j = 0$ if all differences are non-negative. We define

$$K \ominus K' = \#i.(c_1 - c'_1) \ldots (c_{j-1} - c'_{j-1}).c_j. \ldots c_l.$$

In other words, $K \ominus K'$ is the usual vector minus on position offsets but ignores those of K' starting from the first negative difference. The reason for ignoring such position offsets is the same as for $\oplus$. $K \ominus K'$ has the same length as K and degenerates to the ordinary $-$ for flat lists. In Figure 3 we have seen that $\#1.1.0.2 \oplus \#1.3.1 = \#1.4.1$. Interestingly, $\#1.4.1 \ominus \#1.1.0.2 = \#1.3.1$. In general, we have

Theorem 3.1 *For entry keys or search keys K_1, K_2 of the same list, $(K_1 \oplus K_2) \ominus K_1 = K_2$.*

By Theorem 3.1, we can compute the exact position within a leaf node for starting a sequential scan: suppose that we have reached leaf node v and suppose that K_1 has accumulated position offsets for list #i on the left side of v when descending the tree. Let K_2 give the starting position of the sequential scan in terms of the position offsets of #i within v. Then $K_1 \oplus K_2$ is equal to the search key of the list being searched and thus is known. From Theorem 3.1, K_2 is computed by $(K_1 \oplus K_2) \ominus K_1$.

4. Search

A search request is specified by a position path $\#i.s_1 \ldots s_n$ and a positive integer l. This request will search for the list specified by

$$\#i.s_1 \ldots s_{n-1}.s_n,$$
$$\#i.s_1 \ldots s_{n-1}.(s_n + 1),$$
$$\ldots,$$
$$\#i.s_1 \ldots s_{n-1}.(s_n + l - 1),$$

in that order, where searching a list implies returning all its descendent oids that are properly nested. Let $len(P)$ denote the length, i.e., the number of child lists, of the list specified by a position path P. For the above request to be valid, the following conditions should hold: (a) $s_n + l - 1 \leq len(\#i.s_1 \ldots s_{n-1})$, and (b) for all $j < n - 1$, $s_{j+1} \leq len(\#i.s_1 \ldots s_j)$.

Search($\#i.s_1 \ldots s_n, l$). As the first step, the operation searches for the oldest ancestor containing list id #i. Assume that the node is found (otherwise, stop). Initialize the variable $\mathcal{X}$ to $\#i.0$. $\mathcal{X}$ will be used to accumulate position offsets of #i on the left side of the path traversed. The rest of the search is guided by the search key $\mathcal{K}$ of $\#i.s_1. \ldots .s_n$ and $\mathcal{X}$. Let $v.\#j$ denote the list id in entry key K_j in node v. Initially, v is the root. In the following, operators $\oplus$, $\ominus$, and K_i^* are defined as in Section 3.

Step 1 (#i is not in internal node v). Visit non-leaf node v. Let v have entries $< K_1, Pr_1 >, \ldots, < K_p, Pr_p >$. If #i is larger than the largest list id $v.\#p$, the list $\#i$ cannot be found, stop. Assume $\#i \leq v.\#p$. If #i is not contained in v, find k such that $v.\#(k-1) < \#i < v.\#k$, and replace v by the child node under branch Pr_k and goto Step 1 (the non-existing $v.\#0$ is treated as the minimal). If #i is contained in v, goto Step 2.

Step 2 (#i is in internal node v). If $\mathcal{K} > \mathcal{X} \oplus K_p^*$ (note that K_p is the last entry key in v), the list specified by $\#i.s_1 \ldots s_n$ cannot be found, stop; otherwise, find k such that $\mathcal{X} \oplus v.K_{k-1}^* < \mathcal{K} \leq \mathcal{X} \oplus v.K_k^*$. If $v.\#(k-1) = \#i$, $\mathcal{X} \leftarrow \mathcal{X} \oplus K_{k-1}^*$. Replace v by the child node under branch Pr_k. If v is a leaf node, goto Step 3; otherwise, visit node v and repeat Step 2.

Step 3 (leaf node v). Visit leaf node v. Let $\mathcal{K} \ominus \mathcal{X} = \#i.a_1 \ldots a_n$. To locate the first oid to be searched, scan v to the right a_1 \$1 delimiter of list $\#i$, then a_2 \$2 delimiters of list $\#i$, ..., and a_n \$n delimiters of list $\#i$. If there is a shortage or lack of these delimiters in v, the list cannot be found completely, stop. Assume that all these delimiters are found in v. Then starting from the last delimiter \$ found, scan oids to the right, possible on neighboring leaf nodes, until l \$n delimiters of list $\#i$ are found before encountering any \$j delimiter of $\#i$ for $j < n$. If there is a shortage or lack of these markers, the list cannot be found completely.

Example 4.1 Consider $Search(\#2.2.2.1, 2)$ against the P-tree in Figure 2, which searches for the lists specified by #2.2.2.1 and #2.2.2.2, namely q and r. The search key of #2.2.2.1 is $\mathcal{K} = \#2.1.1.0$ and $\mathcal{X} = \#2.0$. At the root A, #2 is found, $\mathcal{X} \oplus K_1^* = K_1 = \#2.0.2$, and $\mathcal{X} \oplus K_2^* = \#2.3$. Since $\mathcal{X} \oplus K_1^* < \mathcal{K} \leq \mathcal{X} \oplus K_2^*$, node C is visited and

$\mathcal{X}$ is changed to #2.0.2. At node C, $\mathcal{X} \oplus K_1^* = \#2.1.2$, so node F is visited and $\mathcal{X}$ remains unchanged. At node F, $\mathcal{X} \oplus K_1^* = \#2.1$, $\mathcal{X} \oplus K_2^* = \#2.1.1$, so leaf node N is visited and $\mathcal{X} = \#2.1$. This value of $\mathcal{X}$ tells that 1 $1 of #2 is indexed on the left side of node N. Finally, $\mathcal{K} \ominus \mathcal{X} = \#2.0.1.0$ means that the scan starts at the first oid after the first $2 in node N. Since $l = 2$, we scan to the right 2 level-3 lists, which requires to access node O. This gives exactly the oids we want, namely, q and r. □

5. Insertion

5.1. Insert into an existing list

An insertion request is specified by Insertion($\#i.s_1 \dots s_n, IN$). It inserts a list contained in IN right before the list specified by $\#i.s_1 \dots s_n$. $\#i$ must be a list id existing in the tree. IN is a sequence of oids such that level-j lists in IN are ended by $(n+j-1). The insertion can be made at any nesting level and may increase the nesting depth of list #i.

Example 5.1 Let us insert the list $[X[Y]Z[W]]$ immediately before oid l in $[[klmn][[op][qr]][stuv]]$ in Figure 2. The user expects the resulting list to be $[[kX[Y]Z[W]lmn][[op][qr]][stuv]]$. The insertion is done by $Insertion(\#2.1.2, IN)$, where IN should contain $X\$2Y\$3\$2Z\$2W\$3\$2$. The operation first searches for the insertion position using the search key $\mathcal{K} = \#2.0.1$. It will follow the path A,B,E,L. On visiting node L, $\mathcal{X} = \#2.0$, so $\mathcal{K} \ominus \mathcal{X} = \#2.0.1$. Then $X\$2Y\$3\$2Z\$2W\$3\$2$ is inserted after the first oid k of list #2. To maintain property P1, K_2 in nodes E,B and K_1 in node A should be changed to #2.0.6 (note that there is an omitted $2 following l), provided that no overflow occurs in node L. □

An important point to note is that only one root-to-leaf path will be affected by insertion of a list, which is the path that leads to the insertion point, such as path A,B,E,L in the above example.

An insertion may cause a leaf node to overflow. If so, new leaf nodes are created and oids are redistributed among the overflowing node and new nodes. Entries for new nodes are then inserted into the parent of the overflowing node, which may cause the parent to overflow, and so on. An insertion therefore may propagate upwards until at some level no overflow occurs. To be precise, for a P-tree of degree (m, M), insertion into an internal node causes an overflow if the number of entries in the node after insertion is more than M. Usually, a degree (m, M) such that $m \leq M/2$ is used. In the following, we consider only the propagation phase because the search phase is the same as in Section 4.

Assume that, for each branch Pr_k of node v that is followed during the search phase, the tuple $< v, k >$ has been pushed onto a global stack S. Let u be the node that overflows, initially being a leaf node, and let $v_1, \dots, v_p$ be new nodes created. Let $p = 0$ if u does not overflow. To maintain P4, each new internal node, except the root, should be filled up by some specified number of entries between m and M, where (m, M) is the degree of the P-tree. $Propagate\_add(u, v_1, \dots, v_p)$ below inserts entries for new nodes $v_1, \dots, v_p$ into the parent of u and propagates insertion upwards.

$Propagate\_add(u, v_1, \dots, v_p)$:

Case 1: $p = 0$, *the insertion into* u *does not cause an overflow.* $< v, k > \leftarrow pop(S)$. Note that u is the kth child of v. If $K(u)$ (as defined in Section 3), is changed by the insertion into u, update the kth entry of v by $K(u)$. Propagate repeatedly such changes to higher levels until it reaches some node v that is either the root or $K(v)$ is not changed.

Case 2: $p > 0$, *the insertion into* u *causes an overflow.* $< v, k > \leftarrow pop(S)$. Update the kth entry of v by $K(u)$. Assume that new nodes $v_1, \dots, v_p$ have addresses $Pr_1, \dots, Pr_p$. Insert new entries $< K(v_1), Pr_1 >, \dots, < K(v_p), Pr_p >$ into node v right after the kth entry of v. Let new nodes $u_1, \dots, u_q$ be created at the level of v, possibly none. Call $Propagate\_add(v, u_1, \dots, u_q)$.

The height of the tree increases when an insertion into the root causes to pop the empty global stack S. In this case, $< new\_root, 1 >$ is first pushed onto S before popping the stack, where new_root is the new root with only one pointer pointing to the current root of the tree. The rest is handled as usual. This may happen a few times, in which case the height of the tree increases a few levels. On the contrast, an insertion in B-tree index can increase the height of the tree by at most 1.

5.2. Insert a fresh list

Inserting a list whose id is not in the tree is fairly similar to inserting child lists into an existing list, with the following constraints and differences. (a) In the position path $\#i.s_1 \dots s_n$, $s_i = 1$ for $1 \leq i \leq n$, and #i must not be found in the search phase. In particular, in the search phase, if $\mathcal{K} > \mathcal{X} \oplus K_p^*$, where K_p is the last entry key in a visited node, the branch Pr_p will be followed, rather than stop like in the search. (b) In the leaf node visited the new list is inserted between the two lists whose ids are closest to #i. The detail is omitted since it is similar to the insertion into an existing list.

6. Deletion

6.1. Delete child lists from a list

A deletion request is specified by Deletion($\#i.s_1. \dots s_n, l$). It deletes all lists specified by

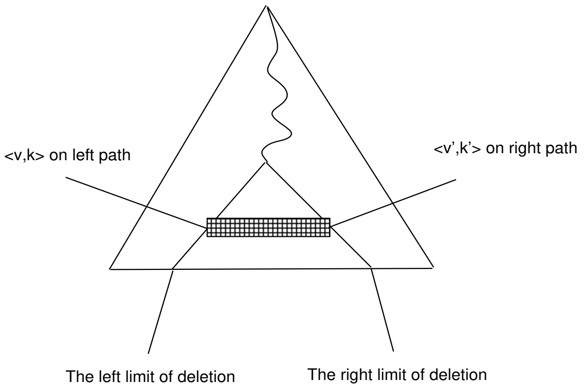

Figure 4. An illustration of deletion

$$\#i.s_1.\ldots s_{k-1}.s_n,$$
$$\#i.s_1.\ldots s_{k-1}.(s_n-1),$$
$$\#i.s_1.\ldots s_{k-1}.(s_n+l-1).$$

To avoid a sequential scan of the nodes to be deleted at each level of the tree, the left and right limits of the deletion are searched by descending along two root-to-leaf paths. Let $\mathcal{K}$ and $\mathcal{K}'$ be the search keys for position paths $\#i.s_1.\ldots s_{n-1}.s_n$ and $\#i.s_1.\ldots s_{n-1}.(s_n+l)$. The search along the left path is guided by $\mathcal{K}$, and the search along the right path by $\mathcal{K}'$. At each level, if branches $v.Pr_k$ and $v'.Pr_{k'}$ are followed, the tuple $< v, v', k, k' >$ is pushed onto the global stack S. Intuitively, $< v, k >$ and $< v', k' >$ encloses all entries (for internal nodes) or oids (for leaf nodes) that will be deleted at this level, shown by the darked band in Figure 4.

Example 6.1 In Figure 2, consider $Deletion(\#2.1, 2)$ to delete the first and second level-1 lists in list #2. The left limit is specified by position path $\#2.1$ and the right limit by position path $\#2.3$. The corresponding search keys are $\mathcal{K} = \#2.0$ and $\mathcal{K}' = \#2.2$. At the root A, due to $\mathcal{K} < K_1^*$ and $\mathcal{K}' < K_2^*$, the left path leads to B and the right path leads to C. $\mathcal{X} = \#2.0, \mathcal{X}' = \#2.0.2$, and $< A, A, 1, 2 >$ is pushed onto the global stack S. In node B, $\mathcal{K} < K_2^*$, so the left path leads to node E and $\mathcal{X}$ remains unchanged; in node C, $\mathcal{X}' \oplus K_1^* < \mathcal{K}' \leq \mathcal{X}' \oplus K_2^*$, so the right path leads to node G and $\mathcal{X}'$ changes to $\#2.1.2$. $< B, C, 2, 2 >$ is pushed onto the global stack S. Then nodes E and G are visited. Both $\mathcal{X}$ and $\mathcal{X}'$ are unchanged and $< E, G, 2, 1 >$ is pushed onto the global stack S. Now $\mathcal{K} \ominus \mathcal{X} = \#2.1 \ominus \#2.0 = \#2.0$ and $\mathcal{K}' \ominus \mathcal{X}' = \#2.2 \ominus \#2.1.2 = \#2.1$. Therefore, in leaf L the first oid following 0 \$1 and 0 \$2 of #2, i.e., k, is the left limit of the deletion; in leaf P, the first oid following 1 \$1 and 0 \$2 of #2, i.e., s, is the right limit of the deletion. All oids from the left limit up to the oid proceeding the right limit, that is, $klmn\$1op\$2qr\$2\$1$, are deleted. An underflow of leaf nodes will cause deletion of nodes from higher levels of the tree. □

Given a deletion request $Deletion(\#i.s_1 \ldots s_n, l)$, let $\mathcal{K}$ and $\mathcal{K}'$ be the search keys for position path $\#i.s_1 \ldots s_{n-1}.s_n$ and position path $\#i.s_1 \ldots s_{n-1}.(s_n+l)$. Let $\mathcal{X} = \mathcal{X}' = \#i.0$ and let S be the empty global stack. The deletion has two phases.

Search phase. Traverse down the tree to the left and right limits of the deletion, guided respectively by $\mathcal{K}$ and $\mathcal{K}'$, as in the search operation. At each level, if branches $v.Pr_k$ and $v'.Pr_{k'}$ are followed, push $< v, v', k, k' >$ onto the global stack S. Assume that eventually leaf nodes v, v', not necessarily different, are reached. Let k be the position of the oid proceeding the last \$ located by $\mathcal{K} \ominus \mathcal{X}$ in v, and , let k' be the position of the oid proceeding the last \$ located by $\mathcal{K}' \ominus \mathcal{X}'$ in v'. The deletion enters the propagation phase by call $Propagate\_del(v, v', k, k'-1)$.

Propagation phase. $Propagate\_del(v, v', i, i')$ will delete from the ith entry or oid of v to and including the i'th entry or oid of v', as shown by the darked band in Figure 4, and propagate deletion upward. However, the deletion does not need to access all nodes deleted. In Figure 4 the nodes in the darked band are deleted by making the left and right ends of the band two consecutive children of a parent node, without actually accessing nodes within the band. Let $K(v)$ be defined as in Section 3.

$Propagate\_del(v, v', i, i')$: First, we assume that v and v' are distinct nodes. $v_1 \leftarrow v$ and $v_2 \leftarrow v'$. $< v, v', i, i' > \leftarrow pop(S)$. Note that v and v' are parents of v_1 and v_2 respectively. Delete all entries or oids starting from and including the ith entry or oid in v, and delete all entries or oids up to and including the i'th entry or oid in v'. Let us consider the following three cases of v_1 and v_2.

Case 1: Neither v_1 nor v_2 underflows. No merging or redistribution is needed. Modify the entry key $v.K_i$ by $K(v_1)$ and modify the entry key $v'.K_{i'}$ by $K(v_2)$. Call $Propagate\_del(v, v', i+1, i'-1)$, where $i+1$ and $i'-1$ are chosen so that the ith entry in v and the i'th entry in v' are not deleted. The same applies to other cases below.

Case 2: Some of v_1 and v_2 underflows but their merging does not. If the merging of v_1 and v_2 does not overflow, merge v_2 into v_1, modify the entry key $v.K_i$ by $K(v_1)$, and call $Propagate\_del(v, v', i+1, i')$. If the merging of v_1 and v_2 overflows, redistribute entries in v_1 and v_2 evenly, modify the entry key $v.K_i$ by $K(v_1)$ and modify the entry key $v'.K_{i'}$ by $K(v_2)$, and call $Propagate\_del(v, v', i+1, i'-1)$.

Case 3: The merging of v_1 and v_2 underflows. This case may occur because oids or entries may be deleted from both v_1 and v_2. Merge or redistribute v_1 and v_2 with one of their neighbor nodes. We omit the detail since there is no implementation difficulty here.

Now let us consider the case that v and v' are identical in the call $Propagate\_del(v, v', i, i')$. Within the call, $v_1 = v_2$. Delete the ith entry or oid up to and including the i'th entry or oid of v_1. If v_1 does not underflow, $< v, v', i, i' > \leftarrow pop(S)$ (note that $v = v'$ and $i = i'$) and propagate the change of $K(v_1)$ upward as in Case 1 of insertion operation. If v_1 underflows, merge or redistribute v_1 with one of its neighbor nodes as in Case 3 above. If v_1 has no such a neighbor, v_1 is returned as the root of the new tree.

6.2. Delete a level-0 list

We treat deleting a level-0 list as the case of deleting all child lists from the list, provided that the list id #i for an empty list is always deleted. This requires to know the length of the list beforehand.

7. B-tree as a special P-tree

It is very interesting to note that the P-tree is in fact a generalization of the traditional B-tree. A record in a relational database can be considered as a flat list of length 1 with the record id being the list id. In this special case, a node entry of the P-tree has the form $< \#i.1, Pr >$, where id #i is a record id. Property P1 states that a record is indexed within the subtree rooted by a node containing the record's id, which holds trivially for the B-tree. Properties P1 and P2 become $\#1 < \#2 < \ldots < \#p$ and $\#(i-1) < \#j \leq \#i$, exactly the requirement of the B-tree on entry keys.

Since a record id appears in at most one entry in an internal node, for the search operation in Section 4, $K_i^* = K_i$ for every entry key K_i, and $\mathcal{X}$ is never changed because $v.\#(k-1) = \#i$ is never true in Step 2. Therefore, $\mathcal{X} \oplus v.K_{k-1}^* < \mathcal{K} \leq \mathcal{X} \oplus v.K_k^*$ in Step 2 can be replaced by $v.K_{k-1} < \mathcal{K} \leq v.K_k$, and $\mathcal{K} \ominus \mathcal{X}$ in Step 3 can be replaced by $\mathcal{K}$. Then it is not difficult to see that the search operation for the P-tree becomes exactly the search operation for the B-tree. The insertion and deletion in the B-tree are special cases of inserting a new length-1 list and deleting a whole length-1 list in the P-tree. This has shown that the P-tree is a natural generalization of the B-tree for handling nested lists.

8. Conclusion

We proposed an index scheme, the P-tree, to support efficient processing of position-related queries and updates on list data. The P-tree index naturally generalizes the conventional B-tree that index flat and length-1 lists, i.e., records, to index nested and variable length lists, i.e., arrays of dynamic cardinality and dynamic dimensions. An alternative view is that while the B-tree provides an index on the inter-object relationship, the P-tree provides an index on both the inter-object relationship (sorted by id's of level-0 lists) the intra-object relationship (sorted by position of level-i lists).

Acknowledgements. We thank Limsoon Wong for helpful discussions on list data and its applications.

References

[1] F. Bancilhon, S. Cluet, C. Delobel. A query language for the O2 object-oriented database system. *DBPL* 1989

[2] M.J. Carey, D.J. DeWitt, J.E. Richardson, E.J. Shekita. Object and file management in the EXODUS extensible database system. *VLDB* 1986, pp. 91-100

[3] M.J. Carey, D.J. DeWitt, S.L. Vandengerg. A data model and query language for EXODUS. *SIGMOD* 1988

[4] A. Guttman. R-trees: a dynamic index structure for spatial searching. *SIGMOD* 1984, pp. 47-57

[5] H-P. Kriegel. Performance comparison of index structures for multi-key retrieval. *SIGMOD* 1984, pp. 186-196

[6] D.E. Knuth. *The art of computer programming*. Volume 3/Sorting and searching, 1973, Addison-Wesley publishing company

[7] J. Nievergelt, H. Hinterherger, K.C. Sevcik. The grid file: an adaptable, symmetric multi-key file structure. *ACM Trans. on Database Systems*, Vol. 9, No. 1, 38-71, 1984

[8] J. Orenstein. Spatial query processing in an object-oriented data system. *SIGMOD* 1986, 326-336

[9] J. Richardson. Supporting lists in a data model. *VLDB* 1992, pp. 127-138

[10] B. Seeger and H-P. Kriegel. *The buddy-tree: an efficient and robust access method for spatial data base systems*. *VLDB* 1990, pp. 59 0-601.

[11] P. Seshadri, M. Livny, R. Ramakrishnan. Sequence query processing. *SIGMOD* 1994, pp. 430-441

[12] M. Stonebraker, H. Stettner, N. Lynn, J. Kalash, and A. Guttman. Document processing in a relational database system. *ACM Tr ans. Office Info. Sys.*, 1, 2, April 1983

[13] B. Subramanian, S.B. Zdonik, T. W. Leung, and S. L. Vandenberg. Ordered types in the AQUA data model. *DBPL*, New York, August 1993, pp. 115-135