P-Tree: A B-Tree Index for Lists

Ke Wang
Beng Chin Ooi
Sam Yuan Sung
Department of Information Systems and Computer Science
National University of Singapore
Lower Kent Ridge Road, Singapore 119260
email: {wangk,ooibc,ssung@iscs.nus.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 set through positioning and nesting elements within the list. The nested position can be specified as either the predicate or the target of a list operation. 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. Based on the known B-tree index, we present an indexing structure, called the P-tree (P for position), to index the position for multiple nested lists. Search and update of lists can be performed at any nesting level and position through the P-tree, and lists can grow and shrink dynamically to an arbitrary nesting depth. Other list-specific operations such as length function and merging or splitting lists are supported in the same index. The P-tree naturally generalizes the B-tree from indexing flat and length-1 records to indexing ordered, nested, variable length lists, and still maintains all performance properties of the B-tree. Experiment results show that the P-tree is efficient and consistent in performance.

Key words. list (or sequence), indexing, update, B-tree, secondary storage

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 [BCD89, GDV88, R92, SLR94, SZLV93] for example). The need of such lists arises from applications that store ordered relations [S83], 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 (i.e., inherent) nature of such data, e.g., disk space structured as ordered cylinders, tracks, sectors, bytes, and bits. See [MV93] for excellent motivating materials on call for order. The second nature of lists is the dynamism of their growth. Lists can grow or shrink in size anywhere and to any level of nesting, not necessarily constrained to occur at the beginning or end of a list. The high frequency of applications involving lists reflects the intuition that list is the "next most" natural data type after set.

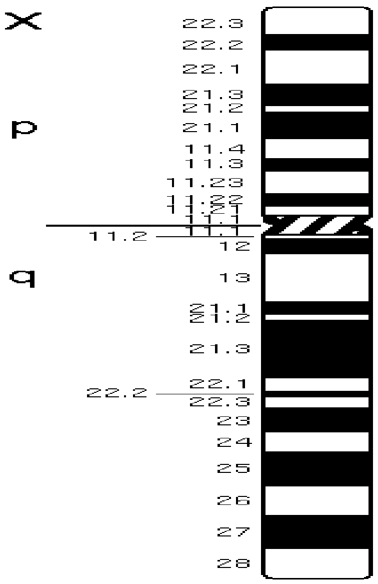


Figure 1: An ideogram of the human X chromosome, from http:// wwwgdb.gdb.org/

The problem. 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, i.e., position-based, or as the search target of the operation, i.e., position-targeted. A genome database is an example where lists are common. In the biomedical and biochemical research, around 3 billion base pairs in the human genome are organized into microscopic units called chromosomes, such as the X chromosome in Figure 1 as visualized by the geneticist or biochemist. The X chromosome is divided into p and q parts, each part is subdivided into regions called bands in a way meaningful to the geneticist or biochemist, 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 number 20 is the id of the chromosome involved. More materials in this area can be found in the Genome Data Base home page *http:// wwwgdb.gdb.org/*.

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. For example, the depth of trees, represented by nested lists, that are generated by programs may have different nesting depths for different inputs, and it is not possible to pre-determine the maximum depth of trees. 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

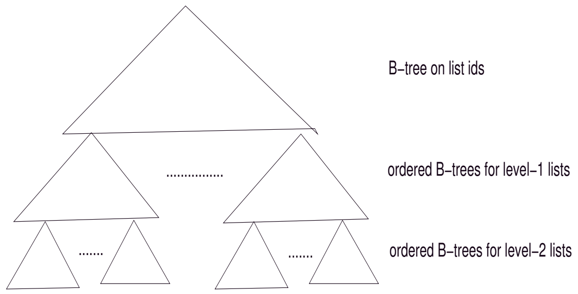


Figure 2: A layered tree index for lists of 2 nesting levels

nesting level as one attribute and apply multi-dimensional indexing methods to such attributes, such as the R-tree [G84], grid file [NHS84], Buddy-tree [SK90], the z-ordering [O86], and the generalized search tree [HNP95]. However, these indexes have not been designed for list data. 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-specific operations such as length function of lists, delete or insert a prefix or suffix, etc.

The second possible approach is to apply the ordered B-tree for a single flat list [CDRS86, K73] 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 [K84], one layer for each nesting level, as shown in Figure 2. 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. This approach, though very simple, loses the essential properties of B-tree and will have poor performance. For the convenience of comparison, let us call this method the *layered tree index*.

There is a great deal of work and implementation on objects with long fields, such as the long field managers in Starburst [LL89] and EOS [B92], with the focus on storage allocation/deallocation, read and write operations, movement of long fields, and recovery mechanism. To our knowledge, there is no work done that specifically looks into the problem of indexing positions of multiple nested lists.

Main results. In this paper, we propose an efficient indexing structure, called the *P-tree* (P for Position), to support position-related operations on list data. In particular, the index allows

- to search or update lists at any position and nesting level, both directly and sequentially, while maintaining the structure of lists correctly.
- to find the first leaf page of interest by following one root-to-leaf path of length no more than $log_b N$, where N is the total number of elements in all lists and b is a "large" number.
- to access the rest of data without accessing unwanted data once the first useful page is found.
- to restrict the change of the index to one root-to-leaf path for search and at most two such paths for update of an arbitrarily large portion of a list.

- lists to be nested dynamically to an arbitrary depth.

A few other features of the P-tree are worth mentioning. First, since all lists in the database are indexed in a single balanced P-tree, the use of the P-tree is justified as long as the total size of all lists is large; the database could consist of either many but short lists, or few but long lists. On the contrast, the layered tree index, essentially one tree per list, requires the list to be sufficiently long, leading to separate access and storage allocation strategies for short and long lists.

Second, entries in a P-tree node can be of variable length. By supporting variable length entries, the number of entries is not affected by few occurrences of long lists, assuming that the majority of lists are of average length. Quite often, there are a large number of flat lists and a small number of deeply nested lists, in which case long entries are used only for deeply nested list data. If all lists are flat and have length 1, the P-tree (plus its operations) degenerates into the B-tree without modification. No assumption is made on the maximum nesting depth of lists.

Third, while a maximum number of immediate sublists within a list is assumed in this paper, it can be lift out by using a variable length field for position number of sublists, in the same manner as for nesting levels. In fact, this assumption is not a severe one because the space of 2 bytes can hold the maximum position number of 65536, thus, allow so many immediate sublists within a list, which is adequate in most practical cases.

Fourth, for position-targeted queries where the search key is some non-positional attribute and the position information is to be retrieved, the P-tree will be used in conjunction with a B-tree on search keys. In this case, the B-tree serves as a secondary index which directs the search to the leaf nodes of the P-tree that contain the elements satisfying the query condition. The nested position information of these elements are found by traversing the P-tree bottom up toward the root. See more discussions on this in Section 3. In this paper, we shall focus on the P-tree index, which plays an essential role in processing both position-based and position-targeted operations.

The paper is organized as follows. Section 2, the main section, presents the data structure of the P-tree and basic operations on the P-tree, namely, search, insertion, and deletion. Section 3 discusses implementation of other useful list operations by the P-tree index. Section 4 examines a compact version of the P-tree. Section 5 presents experimental results on the performance of the P-tree and the compact P-tree under various conditions. Section 6 concludes the paper with additional applications of the P-tree.

2 The P-tree

2.1 Lists

We deal with oids (object ids), not objects themselves. An oid can be either a key value of an object, like the employee number, or a machine generated identifier of an object, like the disk address. If a piece of data is treated as an object, it is considered as atomic and its internal nesting and position structure is invisible to the P-tree. Therefore, if there is a need to retrieve and update the internal structure of a complex object, the object should be treated as a list defined below.

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. An oid may appear more than once in a list and may appear in different lists. A list in the list database is uniquely identified by a list id of the form #i. Note that i alone is not a list id, but an index to distinguish different list ids. Just like oids, #i could be either a value that is semantically meaningful to the user, or a machine generated name of a list. For example, a book is a list of chapters, each chapter is a list of sections, each section is a list of subsections, and each subsection is a list of paragraphs; so a book is represented by a nested oids of paragraphs. A library of books is represented by a list database, with each book being a list and a book identifier, such as the call number, being a list id.

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. The transitivity of the parent/child relationship induces the ancestor/descendant relationship in the usual way. Let l be a list in the list database. The *nesting level* of a descendant list l' in l is defined as follows: the nesting level of l is 0; if l' is a child list of some list l'' of nesting level i, then the nesting level of l' is $i+1$. A level-i list is a list whose nesting level is i. Any descendant list of l can be uniquely identified by the positions of all its ancestors plus its own position within its parent list. The *nesting depth* of list l is the maximum nesting level of all descendant lists of l. A terminal list, i.e., an oid, is allowed to appear at any nesting level, not necessarily at the level of the nesting depth. 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 list $L = [[klmn][[op][qr]][stuv]]$ has 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 Data structure of the P-tree

In this paper, the term B-tree means the B^+-tree. We propose a B-tree based index, called the *P-tree* (P for position), to support efficient search and update of lists. 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 a list id and and $c_i(\geq 1)$ is the *position offset* of a level-i list, explained below. $T(Pr)$ denotes the subtree under branch Pr. Leaf nodes in the P-tree contain oids interleaved with list ids #1,#2, ... and delimiters \$1, \$2, ... In a leaf node, #i marks the beginning of the list #i, and \$j marks the end of a level-j list. Oids from the same list may be stored in multiple leaf nodes, and one leaf node may contain oids of multiple lists in the database. Entry keys may have different length, even for the same list, to allow dynamic growing and shrinking of nesting depth of lists.

In the leaf level, when we say that some number of \$i's are followed by some number of \$j's, it means that *all* these \$i's end before *any* of these \$j's begin. When two entry keys $K = \#i.c_1.\ldots.c_l$ and $K' = \#i'.c'_1.\ldots.c'_{l'}$ 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 $(s_1, \ldots, s_p)$ and $(t_1, \ldots, t_q)$, $(s_1, \ldots, s_p) \leq (t_1, \ldots, t_q)$ 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 $1 \leq i \leq p$.

P-tree. In the *P-tree* of degree (m, M), the following invariants hold for an internal node v with p entries $< K_1, Pr_1 >, \ldots, < K_p, Pr_p >$, where $K_i = \#i.c^i_1 \ldots c^i_{l_i}$:

P1 in the leaf level of subtree $T(Pr_i)$, where $1 \leq i \leq p$, there are exactly c^i_1 \$1's of list #i, which are followed by exactly c^i_2 \$2's of list #i, which are followed by exactly c^i_3 \$3's of list #i, ..., which are followed by exactly $c^i_{l_i}$ $\$l^i$'s of list #i. See Figure 3.

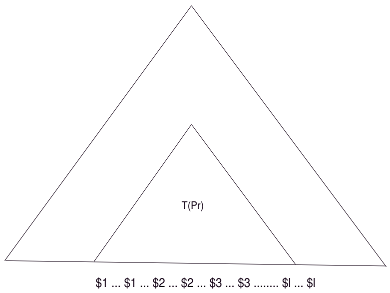


Figure 3: An illustration of P1 on $K = \#.c_1 \dots c_l$. There are c_i \$i's

P2 $\#1 \leq \#2 \leq \dots \leq \#p$. This differs from the conventional B-tree in which $<$ is required. However, whole entry keys K_i's are not necessarily sorted.

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

P2 and P3 together imply that list ids #i are sorted throughout the tree.

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 must be at the same level of the tree.

All leaf nodes of the P-tree are chained to facilitate forward and backward sequential scans. Properties P1,P2, and P3 are required for search and update operations of the P-tree.

Example 2.1 *Consider two lists*

$$L = [[abc][def][ghij]], \textit{ with id } \#1$$

and

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

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 and L′ should be stored as*

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

#2k\$2l\$2m\$2n\$2 \$1o\$3p\$3\$2q\$3r\$3\$2\$1s\$2t\$2u\$2v\$2\$1

in the leaf level of the P-tree. To save space, the following more practical convention is used: 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 4 shows a P-tree of degree $(2, 4)$ *for lists L and L′. In leaf node H, only oid a is ended by \$2; 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 delimiter \$2; 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 of list #2, followed by 2 \$2's of list #2, in that order;* $K_2 = \#2.0.2$ *in node E because in the leaf L there is 0 \$1 (of list #2), followed by 2 \$2's, one for k and one for l;* $K_1 = \#2.1.2$ *in node G because in the leaf P there is 1 \$1 of list #1, followed by 2 \$2, one for s and one 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.*

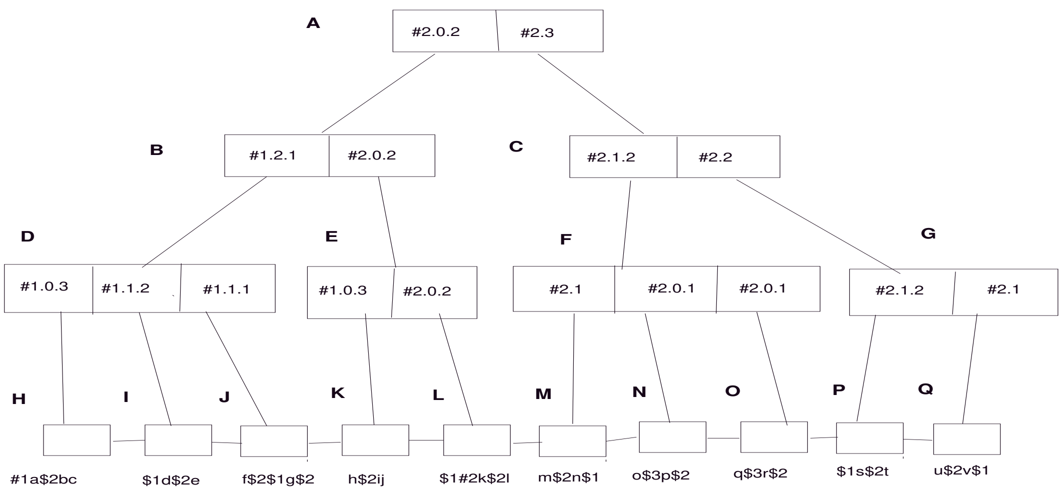


Figure 4: A P-tree structure for list [[abc] [def] [ghij]] and list [[klmn] [[op] [qr]] [stuv]]

Implementation issues. For an entry of the form $< \#i.c_1 \ldots c_l, Pr >$ in the P-tree, the list id #i, position offset c_i, and pointer Pr have fixed size in byte. In our experiment we choose 2 bytes for each c_i, which is sufficient to index up to 65536 immediate child lists within a list. The number l of position offsets in an entry is variable and depends on the nesting depth of the data indexed under this entry. An entry $< \#i.c_1 \ldots c_l, Pr >$ has the stored format in Figure 5. The id #i is always followed by c_1. A c_i field could be followed by either c_{i+1} or the special symbol & (of fixed size) that marks the end of entry key $\#i.c_1 \ldots c_l$. One additional bit at the end of each c_i field is used for distinguishing these two possibilities: 1 indicates that c_{i+1} follows and 0 indicates that & follows. & is always followed by the Pr field. Since the format of the next field can always be derived from the previous one, all fields can be decoded.

There are two approaches to mapping an internal P-tree node to disk storage, depending on whether the page size is fixed or variable. In the fixed page size approach, a maximum value of the length l is imposed, but no uniform value of l is required for all entries. In the variable page size approach, an internal node in the P-tree is implemented as a logical page consisting of one or more disk blocks. No maximum value of the length l is imposed in this case. This approach is particularly attractive if the maximum nesting depth of lists is not known beforehand and lists can be dynamically nested to an arbitrary depth. In most practical cases, by tuning the degree (m, M) of the P-tree almost every node requires no more than one disk blocks of storage space. Variable size entries were previously used in the Cell-tree [G], an index for geometric databases.

A leaf node is implemented by one disk block. For the case of known maximum nesting depth, a delimiter $i, where i is no more than the maximum nesting depth, will have fixed size in byte. For the case of unknown maximum nesting depth, a delimiter $i will have variable size. We assume that all oids have fixed size in byte. A variable size delimiter can be stored as a sequence of segments, where each segment is one byte. In a leaf node, a delimiter segment could be followed by the next segment of the same delimiter, or the first segment of another delimiter, or an oid, or a list id #j. Similarly, a leaf node can begin with any of these four fields. An oid could be followed by either an oid or a delimiter segment. Therefore, by adding one bit to an oid field, two bits to a delimiter segment, and allocating two bits in the beginning of a leaf node, the format of the next field can always be determined.

From P2 and P3, we have the following fact about distribution of list ids in the P-tree.

#i	c1	1	c2	1	....	cl	0	&	Pr

Figure 5: The stored format of entry $< \#i.c_1 \ldots c_l, Pr >$

Fact. If a P-tree contains list id #i, all internal nodes containing #i forms a subtree, therefore, share the common oldest ancestor containing #i, called the subroot of #i below.

As the subroot of #i is always visited in search for any data in list #i, a practical implication of this fact is that, if any information of a list #i such as length, size, nesting depth is stored with the subroot for #i, it can be updated with little cost. In implementation, this can be done by storing a pointer in the subroot that points to an area containing such information.

2.3 Specifying lists

In this section, we define a mechanism for specifying the portion of a list to be searched or updated by the means of position paths defined below.

Position path. The first list to be searched is specified by a *position path* of the form $\#i.s_1 \ldots s_n$, where $s_i > 0$ and $n > 0$, which 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 list #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 $\#i$. However, in search of the P-tree, it is the entry key format of this position path that is actually used, where the relative position within a list is expressed in terms of "follow" rather than "within". 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 $\#i.4.6.2$. In general, we have

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

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

2.4 Two housekeeping operators

To determine the current position within a list while searching or updating the P-tree, we need two housekeeping operators on entry keys and search keys. Consider two entry keys or search keys $K = \#i.c_1 \ldots c_l$ and $K' = \#i.c'_1 \ldots c'_{l'}$ of the same list. 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 latest \$(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 $+$ on the single position offset 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 6 (all \$i's refer to those of #1): if subtree T1 contains 1 \$1 followed by 0 \$2 followed by 2 \$3's, and if subtree T2, which is immediately on the right of T1, contains 3 \$1's followed by 1 \$2, then T1 and T2

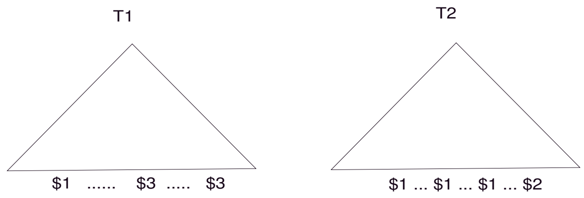


Figure 6: An illustration of $\#1.1.0.2 \oplus \#1.3.1 = \#1.4.1$

together contain 4 \$1's followed by 1 \$2.

The operator $\oplus$ allows us to accumulate position offsets (of the list being searched) in the subtrees under an internal node, therefore, accumulate position offsets on the left side of a path traversed. To see this, let $K_i = \#i.c_1^i.\ldots.c_{l^i}^i$, where $1 \le i \le p$, be all entry keys in an internal node v. We define K_i^* as

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

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's of list #i, followed by exactly a_2 \$2's of list #i, ..., followed by exactly a_q \$d's. 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's of list #i, followed by exactly c_2 \$2's of list #i, followed by exactly c_3 \$3's of list #i, ..., followed by exactly c_l \$l's 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.

Theorem 2.1 *Property P1 holds if and only if $v.K_i = K(v_i)$ for all internal nodes v and children v_i.*

This computation of $v.K_i$ is similar to the B-tree, that is, derived from the last entry key in the ith child of v. The difference is that the P-tree will accumulate position offsets from the first to the last entry in the ith child. As the idea of Theorem 2.1 is intuitive, the proof is omitted here.

Example 2.2 *Consider the tree in Figure 4. In node D, $K_1^* = \#1.0.3$ and the leaf node H contains 0 \$1 and exactly 3 \$2's of list #1 (2 \$2's not shown); $K_2^* = \#1.1.2$ and node I contains exactly 1 \$1 of list #1 followed by exactly 2 \$2's of list #1 (1 \$2 not shown); $K_3^* = \#1.1.1$ and node J contains exactly 1 \$1 followed by 1 exactly \$2 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's of list #1 followed by 1 \$2 of list #1; $K_2^* = \#2.0.2$, so in the subtree rooted at node E there are 0 \$1 of list #2, followed by 2 \$2's 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.*

We now define the second housekeeping operator on entry keys. Let $K = \#i.c_1 \ldots c_l$ and $K' = \#i.c'_1 \ldots c'_{l'}$ be two entry keys or search keys of the same list. 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 6 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 2.2 *For entry keys or search keys K_1, K_2 of the same list, $(K_1 \oplus K_2) \ominus K_1 = K_2$.*

A formal proof of Theorem 2.2 can be found in the technical report of this paper. By Theorem 2.2, 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, see subsection 2.5 below. 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 2.2, K_2 is computed by $(K_1 \oplus K_2) \ominus K_1$.

2.5 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

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

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 subsection 2.4.

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's of list $\#i$, then a_2 \$2's of list $\#i$, ..., and a_n \$n's 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 neighbouring leaf nodes, until l \$n's 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 2.3 *Consider* $Search(\#2.2.2.1, 2)$ *against the P-tree in Figure 4, 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*.*

In the following discussion, we assume that there is a storage manager that allocates and deallocates disk blocks on request. In particular, when a new node is created or an existing node is deleted, the manager will handle the allocation and deallocation of disk space. This includes the mapping between the concept of page used by the P-tree and the concept of block at the physical implementation level. Having assumed this, we will focus on the algorithmic aspect of insertion and deletion operations without mentioning the underlying storage management.

2.6 Insertion

Two kinds of insertions are considered, namely, insertion into an existing list and insertion of a fresh list.

2.6.1 Insert into an existing list

This refers to inserting new child lists into an existing list. An insertion request is specified by Insertion($\#i.s_1 \ldots s_n, IN$). It inserts a list contained in IN right before the list specified by $\#i.s_1 \ldots s_n$. $\#i$ 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 2.4 *Let us insert the list* $[X[Y]Z[W]]$ *immediately before oid* l *in* $[[klmn][[op][qr]][stuv]]$ *in Figure 4. 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.*

Some housekeeping work is needed to translate between the user's view of lists and their leaf representation. 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. This follows as a consequence of property P1, that is, an entry key depends only on the subtree under its branch. It is this property that allows dynamic change of nesting levels of lists along the affected path without changing the rest of the tree.

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, and deletion from an internal node causes an underflow if the number of entries in the node after deletion is less 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 subsection 2.5.

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 $v_1, \ldots, v_p$ 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. $Propagate\_add(u, v_1, \ldots, v_p)$ below inserts entries for new nodes $v_1, \ldots, v_p$ into the parent of u and propagates insertion upwards.

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

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 Theorem 2.1, 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 $v_1, \ldots, v_p$ have addresses $Pr_1, \ldots, Pr_p$. Insert new entries $< K(v_1), Pr_1 >, \ldots, < K(v_p), Pr_p >$ into node v right after the kth entry of v. Let new nodes $u_1, \ldots, u_q$ be created at the level of v, possibly none. Call $Propagate\_add(v, u_1, \ldots, u_q)$.

Note that 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.

2.6.2 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 $\#i.s_1 \ldots s_n$, $s_i = 1$ 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.

2.7 Deletion

Similar to insertion, we have two kinds of deletions, namely, delete some portion of a list and delete a whole list.

2.7.1 Delete child lists from a list

A deletion request has the form Deletion($\#i.s_1 \ldots . s_n, l$). It deletes all lists specified by

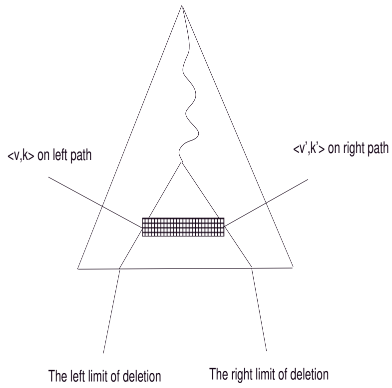


Figure 7: An illustration of deletion

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

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 $\#i.s_1.\dots s_{n-1}.s_n$ and $\#i.s_1.\dots s_{n-1}.(s_n+l)$. 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 7.

Example 2.5 *In Figure 4, 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 $Deletion(\#i.s_1 \dots s_n, l)$, let $\mathcal{K}$ and $\mathcal{K}'$ be the search keys for position path $\#i.s_1 \dots s_{n-1}.s_n$ and position path $\#i.s_1 \dots s_{n-1}.(s_n+l)$. 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 7, and propagate deletion upward. However, the deletion does not need to access all nodes deleted. In Figure 7 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 Theorem 2.1.

$Propagate\_del(v, v', i, i')$: 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 $Propagate\_del(v, v', i, i')$. Note that 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.

2.7.2 Delete a whole list

To delete a whole list, one approach is to treat it as the special case of deleting all child lists from the list, provided that the list id #i for an empty list is always deleted. But this requires to know the length of the list beforehand. Alternatively, we descend the tree to search for the two limits of the list, where the right limit refers to the beginning of the list following the end of list #i in the leaf level, or the end of the leaf chain if no such a list exists. If a visited node does not contain the id #i of the delete list, branches are followed as in subsection 2.7.1. Assume that a visited node v contains id #i. Let Pr_l be the first branch for #i in v. Let Pr_r be the last branch in v if the last branch is for #i, or else let Pr_r be the branch immediately on the right of the last branch for #i in v. Then follow branches Pr_f and Pr_l. The rest of the operation is the same as in subsection 2.7.1.

2.8 B-tree as a special P-tree

A tuple in a relational database can be considered as a flat list of length 1 with the tuple id being the list id. In this special case, a node entry of the P-tree has the form $< \#i.1, Pr >$, where the tuple id #i appears in at most one entry in an internal node. Property P1 states that a record is indexed within the subtree rooted by a node containing the record's id, trivially true for the B-tree.

The conditions in P1 and P2 become $\#1 < \#2 < \ldots < \#p$ and $\#(i-1) < \#j \leq \#i$, exactly the requirement of the B-tree on entry keys. P4 and P5 are also required in the B-tree. In this case, the only difference between the P-tree and the B-tree is the redundant 1 in entry key #i.1 and the extra right-most entry key in each node that is omitted in the B-tree.

Since a tuple id appears in at most one entry in an internal node, for the search operation in subsection 2.5, $K_i^* = K_i$ 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. Therefore, these P-tree operations degenerate to insertion and deletion on the B-tree. This has shown that the P-tree is a natural generalization of the B-tree for handling nested lists.

3 Other list operations

In addition to the basic search, insertion, and deletion of lists, other list operations often arise as well in real applications and need to be supported efficiently. With some minor extensions of the P-tree, these operations can be implemented much more efficiently.

Find the beginning of a list. It will point the current access position to the first oid of a list specified by a position path, for the seek of subsequent sequential processings. The algorithm is similar to the search operation with parameter $l = 1$, but no data will be returned.

Find the end of a list. This searches for the end of a list specified by a position path. See the length function below.

Length function. Given a position path $P = \#i.s_1 \ldots s_n$, the length function returns the number of child lists in the list specified by P, i.e., $len(P)$. We search for the right limit of P, guided by the search key $\mathcal{K}$ of position path $\#i.s_1 \ldots s_{n-1}.(s_n+1)$. For this to be correct, a dummy child list is conceptually appended as the last child list in every list (but it is not visible in other operations). Assume that a leaf node L is reached. Let $\mathcal{X} = \#i.a_1 \ldots a_p$ and $\mathcal{K} \ominus \mathcal{X} = \#i.b_1.\ldots.b_n$. Let node L contain x \$(n+1)'s of list #i between the (b_n-1)th and b_nth \$n of #i. Intuitively, x is the number of child lists of the list specified by P in L. If $p \leq n$ or $a_1.\ldots.a_{n-1}.a_n \neq s_1.\ldots.s_{n-1}.s_n$, return x; otherwise, a_{n+1} is the number of child lists of P appearing on the left of L, so return $x + a_{n+1}$.

Example 3.1 *Let us compute $len(\#2.2)$ using the P-tree in Figure 4. Since #2.2 has two child lists, [op] and [qr], we expect the length function to return 2. In this case, $n = 1$, $s_1 = 2$, and the search key $\mathcal{K} = \#2.2$ (for position path #2.3). Nodes A, C, G are visited in sequence. At node G, $\mathcal{X} = \#2.1.2$ (i.e., $a_1 = 1$ and $a_2 = 2$) and $K_1^* = \#2.2.0.2$, so leaf P is visited. Then $\mathcal{K} \ominus \mathcal{X} = \#2.1$ (i.e., $b_1 = 1$). Since node P contains no \$2 of #2 before the first \$1 of #2, $x = 0$, indicating that no child list of the list specified by position path #2.2 appears in node P. Thus, $x + a_2 = 2$ is returned as $len(\#2.2)$.*

Append at the beginning of a list. This is done by $Insertion(\#i.s_1 \ldots s_{n-1}.1, IN)$, which inserts a list before the first child list of the list specified by position path $\#i.s_1 \ldots s_{n-1}$.

Append at the end of a list. This is similar to finding the end of a list as above. Once the end is found, insertion and propagation are performed as in the insertion into a list.

Delete a prefix of a list. This is done by $Deletion(\#i.s_1 \ldots s_{n-1}.1, l)$, which deletes the given number l of child lists of the list specified by position path $\#i.s_1 \ldots s_{n-1}$, starting from the first child.

Delete a suffix of a list. Descend the tree along a left path and a right path. The left path is guided by the starting position of the suffix, and the right path is guided by the end of the list, as in finding the end of a list. After reaching leaf nodes, deletion and propagation proceed as in the basic deletion.

Merge two lists. This is done by deleting the shorter list and appending it to the beginning or end of the longer list, depending on the user specification.

Split a list into two. This is done by deleting a prefix or suffix of the list and inserting the deleted prefix or suffix back into the tree under a new list id.

Position-targeted operations. As mentioned in the Introduction, the position information could be the target of a retrieval. In such cases, the search value is on some non-positional attribute and a secondary index on this attribute will map the search value to some leaf nodes of the P-tree that contains the elements satisfying the search condition. To find the position of these elements in their lists, we traverse from these leaf nodes bottom up toward the root of the P-tree and accumulate position offsets along the way. The following example illustrates the idea.

Example 3.2 *Suppose that we have reached leaf node N in Figure 4 from a secondary index on a non-positional attribute, and suppose that we want to find out the position information of the satisfying oid p. By visiting the parent F, we know p is an oid of list #2, and the position offset of p within the subtree rooted by F is computed by* $K_1^* \oplus \#2.0.0.1 = \#2.1.0.1$*, where #2.0.0.1 is the position offset of oid p within N and can be found by accessing node N. By visiting the parent C, we find that the position offset of p in the subtree rooted by C remains #2.1.0.1. Then we visit the root A and the position offset of p in the whole list #2 is computed by* $K_1^* \oplus \#2.1.0.1$*, still #2.1.0.1. The position path format for #2.1.0.1 is #2.2.1.2, which is returned as the position information of p that satisfies the condition on the non-positional attribute.*

4 The compact P-tree

In the P-tree, a long list could span over multiple entries and even multiple nodes in the P-tree. Repeatedly storing the id #i for such lists in every entry is a waste of disk space and overhead of disk access. We now consider methods of reducing the storage space of list ids.

We examine a variation of the P-tree, called the *compact P-tree*, in which a list id is stored at most once in each node.

Consider a list with id #i and the entry keys $K_1, \ldots, K_k$ for #i in some tree node. $K_1, \ldots, K_k$ must be adjacent in the node from property P2 in subsection 2.2. To avoid duplication of #i in these keys, we store #i in the first key K_1 and store only position offsets in $K_2, \ldots, K_k$. These two formats can be distinguished by adding a prefix of 0-1 bit in front of each entry key. Let us call an entry with a list id an *1-entry* and an entry without a list id a *0-entry*. An internal node always starts with an 1-entry of some list, followed by zero or more 0-entry of that list, then an 1-entry of another list, followed by zero or more 0-entry of that list, and so on. Since the length of entry keys is always stored, the list id for a 0-entry can be recovered from a left-to-right scan of proceeding 1- or 0-entries in the node.

Figure 8 shows a compact P-tree of the P-tree in Figure 4, in which 1-entries are those with #i and 0-entries are those without #i. In Figure 8, all entries are kept in the original nodes of Figure 4 for easy correspondence. An actual compact P-tree could be quite different from the original

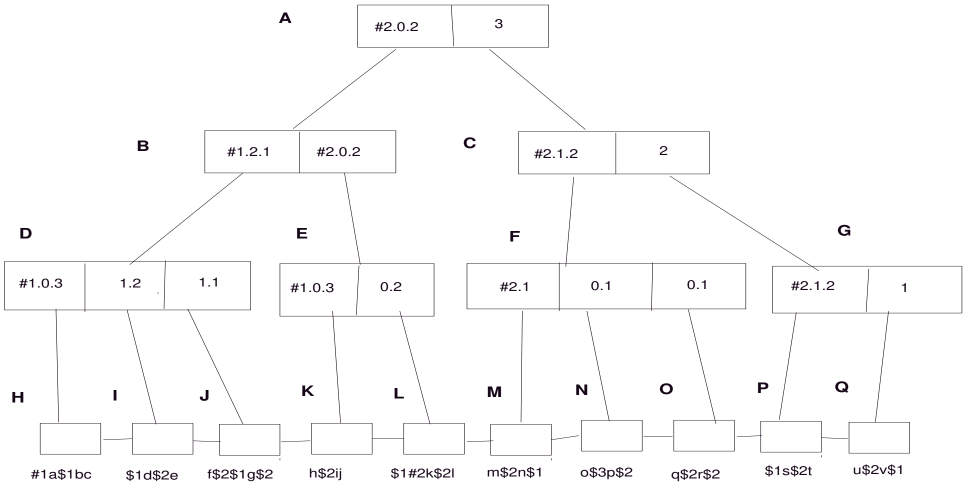


Figure 8: An example for the compact P-tree

P-tree, for example, having a smaller height, because some nodes may contain more entries due to the compactness. For long lists the compactness will reduce the space for entries and increase the branching factor of the tree. For short lists, however, it is likely that all entries in a node are for different lists and are 1-entries. In this case, a p entry node will have an overhead of p bits for storing the 1 and 0 prefixes.

The above idea of compactness can be carried even further to avoid the duplication of list ids across different nodes. In this method, all entries except the first in a node are compacted as before. For a parent node v and its ith child v_i, if the list id for the ith entry in v is the same as the first list id in v_i, the first id in v_i need not be stored. The same applies to the parent of v, the grandparent of v, and so on. The idea is that, in the search or update operation, the parent v has to be visited before the child v_i and the first list id in v_i, if not stored, can be derived from entries in v, which are either stored there or have been derived from the parent of v in the same manner. As a result, for a long list #i spanning over multiple nodes in some level l of the tree, only the first entry for #i in level l is 1-entry; all other entries for #i in that level are 0-entries. In Figure 8, for example, the id #2 in nodes C,F,G and id #1 in node E need not be stored because they can be referred from their parent nodes. However, this method cannot be used if there is a need to traverse the tree bottom up, as is the case for position-targeted operations discussed in Section 3.

Another practical consideration is to allocate a different number of bytes to a position offset at different levels of the tree, based on the observation that nodes at lower levels of the tree will root smaller subtrees and thus have fewer \$ delimiters in leaf nodes of these subtrees. This will effectively increase the branching factor of nodes as there are much more nodes at low levels than high levels.

5 Performance evaluation

5.1 Methodology

To verify the efficiency of the P-tree, the compact P-tree and the layered tree discussed in the Introduction, we implemented all three in C on IBM RS/6000 Model 560 and conducted various experiments. We have chosen the compact P-tree that removes duplicated storage of list ids within

Pointer size	12 Bytes
List id (i.e., #i) size	12 Bytes
Position offset size	2 Bytes
Oid (i.e., a, b, c) size	20 Bytes
Page size	1K or 4K Bytes
Delimiter (i.e., $i) size	1 byte
Minimal page utilization (except the root)	50%
Default page utilization (except the root)	70%
Number of oids	1000000
Number of lists	1000 to 10000
Nesting depth	1 to 10

Figure 9: Parameters used in experiments

a node, but not across nodes. In these experiments, we used the parameters summarized in Figure 9. Since the maximum nesting depth is known beforehand for the used data sets, a fixed page size is used. In Figure 9, the default page utilization is the fill percentage of pages used when allocating storage for a list requiring more than one page.

We focus on the height of trees for the following reasons. First, the height of a tree measures the cost of locating the first data page of interest, that is, the search cost. Second, the height reflects the utilization of nodes since an under-utilized tree tends to have more levels. For the same reason, the height also reflects the compactness or the branching capacity of tree nodes. So we present the height of the indexes as an indicative measure for their performance. For the unbalanced layered tree, the height should refer to the weighted length of all root-to-leaf paths, where the weight will reflect the tested access patterns of lists. See more discussions below.

Experiment I studies the effect of nesting depth of lists on the I/O cost, and Experiment II studies the effect of the number of lists while fixing the number of oids in the database. In these experiments, we model both uniformly and non-uniformly distributed list size, using Zipf-distribution with skew factors 0 and 1 [Z49], and model uniformly distributed list length in all nesting levels. Refer to subsection 2.1 for the definition of list size and list length. All terminal lists, i.e., oids, are nested at the maximum nesting level. For example, a list of nesting depth 3 and size 64 will have 4 (level-1) child lists, each of which has 4 (level-2) child lists, each of which contains 4 (level-3) oids. These decisions are in favor of the layered tree index in the sense that they make the layered tree index more balanced than other cases.

The height of the P-tree and the compact P-tree depends on the size of the database, not on the size of individual lists being searched or updated. The height of the layered tree, however, does depend on the size of the list being searched because there is a dedicated tree index for each list. For this reason, we consider two kinds of access patterns of lists. In the *size-uniform access*, the probability of accessing a list is proportional to the size of the list, i.e., the number of oids. In the *list-uniform access*, every list has the equal probability of being accessed, independent of its size. For the layered tree, the height refers to the average height weighed by the probability of accessing lists of different nesting levels. These access patterns make a difference only for non-uniform list size. In addition, the layered tree is used only for long lists that cannot be contained in one page; for short lists that can be contained in a single page, they are indexed only by the top layer B-tree, that is, no layers of ordered B-trees are maintained.

5.2 Experiment I: the effect of nesting depth

The nesting depth of lists affects all three indexes. For the nesting depth of nd, an entry in the P-tree and compact P-tree has nd position offsets, and the layered tree has nd layers of ordered B-trees. Intuitively, the larger nd is, the smaller the branch factor of the P-tree and compact P-tree is, and the taller all three trees are. In this experiment, we vary the nesting depth of lists from 1 to 10, with the total size of lists fixed at 1 million (oids) and the number of lists fixed at 1000. Both uniform and non-uniform distributions of list size are tested. All trees are constructed by random insertion of lists whose size was generated using the Zipf-distribution described above. The average height is given in Figure 10 for page size of 1K and 4K. For the non-uniform distribution of list size, i.e., (a) and (c), the height of the P-tree and compact P-tree is not affected by the access patterns because these trees are balanced. The layered tree index has a larger height for size-uniform access than for list-uniform access because the former gives more weights to larger lists that have deeper trees. This difference of the layered tree index disappears for uniformly distributed list length in (b) and (d) because the trees for all lists have the same depth.

The main findings of Experiment I (Figure 10): under the tested conditions of nesting depth, page size, list size distributions, access patterns, both the P-tree and compact P-tree indexings outperform the layered tree index. The performance gain gets more substantial for deeper lists and smaller page size. Even for flat lists, because the layered tree has to maintain one layer of B-tree and one layer of ordered B-trees, it still marginally loses to the P-tree and compact P-tree. The compact P-tree performs slightly better than the P-tree.

5.3 Experiment II: the effect of the number of lists

This experiment studies the effect of the number of lists on height, while fixing the total size of the database at 1 million oids and the nesting depth at 4. We vary the number of lists from 1000 to 10000. Both uniform and non-uniform list sizes are tested. The results are given in Figure 11 for page size of 1K and 4K. Given the fixed total size, the more lists there are, the shorter they are on average. As expected, the number of lists have not affected much the height of the P-tree because the total size of lists is fixed. The height of the P-tree is increased by 1 only when the number of lists increases from 7000 to 8000, shown in (d). As for the compact P-tree, in theory, short lists could reduce its effectiveness; in the experiment, however, the effectiveness has not been affected much. The P-tree and the compact P-tree have almost the same height, except in the case of (d) for 8000 lists where the compact P-tree stays at height of 3 but the P-tree has increased to 4.

On the other hand, the number of lists has more effects on the layered tree index. First of all, the height of the top B-tree on list ids depends on the number of lists. Secondly, since lists that are short enough to be stored in one page are indexed directly by the B-tree without the layered index, the number of lists determines the size of lists and thus the choice between the non-layered index and the layered index. The combination of these has affected the height in either ways. For very large number of lists, many short lists are indexed by the B-tree directly without using the layered tree index. This will lead to the decreased height for the case of large number of lists. For instance, in Figure 11(d) there is a sharp drop in height at 7000 and onwards because of the switch from the layered index to the B-tree index.

The main findings of Experiment II (Figure 10): under all tested conditions of nesting depth, page size, list size distributions, access patterns, the P-tree and compact P-tree indexes outperform the layered tree index. The performance gain is more substantial for long lists than for short lists. The layered tree index performs better with list-uniform access than size-uniform access, especially for longer lists, as in (a) and (c). The experiment also shows that the P-tree and compact P-tree perform almost equally well, with compact P-tree having a marginal advantage.

5.4 Storage usage

Besides the height of the tree, we also examined the storage usage in both Experiments I and II. Figure 12(a)(b) shows the storage usage in Experiment I, and Figure 12(c)(d) shows the storage usage in Experiment II, both for the non-uniform list size distribution. The case of uniform list size distribution is similar. First, we observed that the P-tree and compact P-tree have very similar storage usage, with the compact P-tree being slightly better. This is a bit of surprise to us as we expected that the compact P-tree could save more storage. On the bright side, it suggests that the pure P-tree is competitive in storage as well. Second, the layered tree index uses much more space than the P-tree and compact P-tree. This is not surprising at all if we recall that the layered tree index maintains one ordered B-tree for each list nested at each level, which will result in too many under-filled roots of the ordered B-trees. It is possible to reduce the storage space by packing several roots into a single node, but entries of some roots have to be stored in more than one node. As a result, the access cost and complexity of the implementation may increase.

6 Conclusion

The high frequency of applications involving large, ordered, nested lists calls for both language and access supports. The main departure of list from set is the addition of the ordering or position of elements within a list. The expressive power of languages can be enhanced by incorporating the position attribute of list data. In this paper, 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 from flat and length-1 lists, i.e., records, to nested and variable length lists, or arrays of dynamic cardinality and dynamic dimensions. When all lists are flat and have length one, the P-tree degenerates to the B-tree without modification. While the B-tree provides an index on the inter-object ordering among simple objects, the P-tree provides a single index on both the inter-object ordering and the intra-object ordering for nested lists. As our experiments have shown, a single index for both kinds of orderings provides not only uniformity, but also substantial performance advantages.

Instead of storing the position information in a node entry, one can store other statistical information about the data being indexed. For example, an entry key K can have the form $\#i.n$, where #i is an usual index key in the B-tree and n is the number of records that are indexed in the subtree under branch Pr. In this one-dimension case, the proposed operators $\oplus$ and $\ominus$ become the usual plus $+$ and minus $-$, and one can answer statistical queries like "what is the size of the result of $\sigma_{K=10}(R)$" in one descending of the P-tree, or "what is the size of the result of $\sigma_{2\leq K\leq 10}(R)$" in two descendings of the P-tree. The ability to answer such queries efficiently is important for estimating the selectivity and size of relational expressions for query optimization, an active research topic in the database field, e.g., [LNS90].

The P-tree index also has application in substring search where strings could be dynamically updated. In the context of substring search, one would like to find all positions in a long string in which a given substring can be found. The suffix tree [M76] contains these positions in the nodes to which the traversing from the root spells out the substring. If the string is updated, it is extremely inefficient to update all affected positions because they could be anywhere in the suffix tree. The P-tree index provides a solution to this problem: the long string is indexed by a P-tree, and the suffix tree contains the addresses of the P-tree leaf nodes starting the substring, rather than the positions of the substring. To find all positions of a given substring, the suffix tree is searched first to find relevant leaf nodes of the P-tree. Then the positions of the substring are found by traversing from these leaf nodes toward the root of the P-tree, as for the position-targeted operation discussed in Section 3. The suffix tree and P-tree are currently studied by the authors as a mechanism for facilitating data mining in sequence data.

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

References

[B92] A. Billiris, "The performance of three database storage structures for managing large objects", SIGMOD 92, pp. 276-285

[BCD89] F. Bancilhon, S. Cluet, C. Delobel, "A query language for the O2 object-oriented database system", 2nd Int'l workshop on Database Programming Languages, 1989

[CDRS86] 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

[G] O. Gunther, "Efficient structures for geometric data management", Lecture Notes in Computer Science, 337 (Edited by G. Goos and J. Hartmanis), Springer-Verlag

[GDV88] M.J. Carey, D.J. DeWitt, S.L. Vandengerg, "A data model and query language for EXODUS", ACM SIGMOD 1988

[G84] A. Guttman, "R-trees: a dynamic index structure for spatial searching", ACM SIGMOD 1984, pp. 47-57

[HNP95] J.M. Hellerstein, J.F. Naughton, A. Pfeffer, "Generalized search trees for database systems", VLDB 95, pp. 562-573

[K84] H-P. Kriegel, "Performance comparison of index structures for multi-key retrieval", ACM SIGMOD 1984, pp. 186-196

[K73] D.E. Knuth, "The art of computer programming", Volume 3/Sorting and searching, 1973, Addison-Wesley publishing company

[LL89] T.J. Lehman and B.G. Lindsay, "The Starburst long field manager", VLDB 1989, pp. 375-383

[LNS90] R.J. Lipton, J.F. Naughton, and D.A. Schneider, "Practical selectivity estimation through adaptive sampling", ACM SIGMOD 90, pp. 1-11

[M76] E.M. McCreight, "A space-economical suffix tree construction algorithm", JACM, Vol. 23, No. 2, April 1976, 262-272

[MV93] D. Maier and B. Vance, "A call for order", PODS 93, pp. 1-16

[NHS84] 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

[O86] J. Orenstein, “Spatial query processing in an object-oriented data system”, SIGMOD 1986, 326-336

[R92] J. Richardson, “Supporting lists in a data model”, VLDB 1992, pp. 127-138

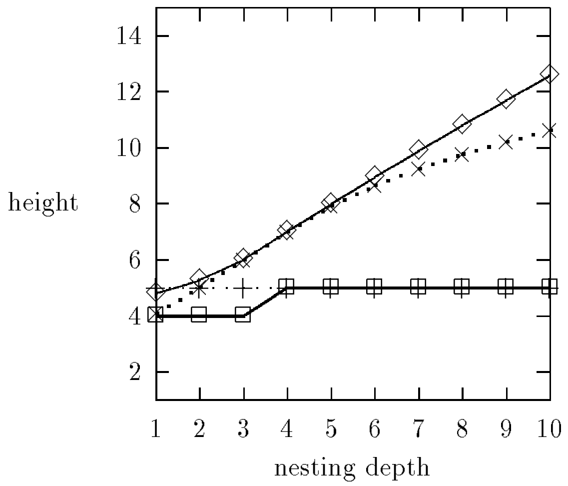
[SK90] B. Seeger and H-P. Kriegel, “The buddy-tree: an efficient and robust access method for spatial data base systems”, VLDB 1990, pp. 590-601.

[SLR94] P. Seshadri, M. Livny, R. Ramakrishnan, “Sequence query processing”, ACM SIGMOD 1994, pp. 430-441

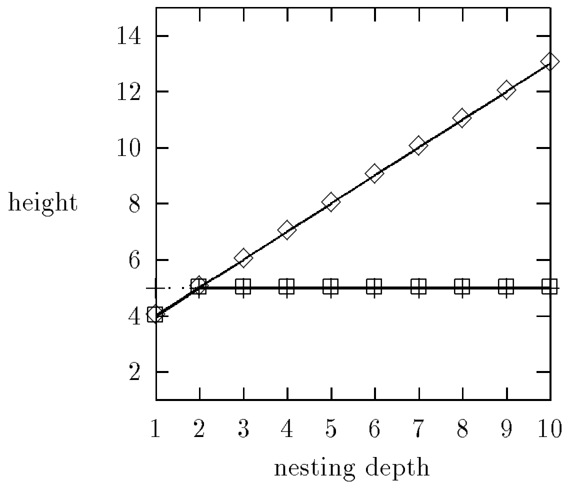
[S83] M. Stonebraker, H. Stettner, N. Lynn, J. Kalash, and A. Guttman, “Document processing in a relational database system”, ACM Trans. Office Info. Sys., 1, 2, April 1983

[SZLV93] B. Subramanian, S.B. Zdonik, T. W. Leung, and S. L. Vandenberg, “Ordered types in the AQUA data model”, Proceedings of 4th International Workshop on Database Programming Languages, New York, August 1993, pp. 115-135

[Z49] G.K. Zipf, *Human behavior and the principle of least effort*, Addison Wesley, 1949

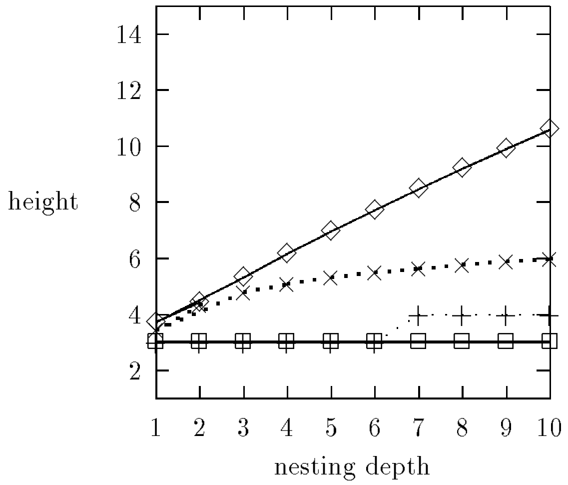


(a) non-uniform size

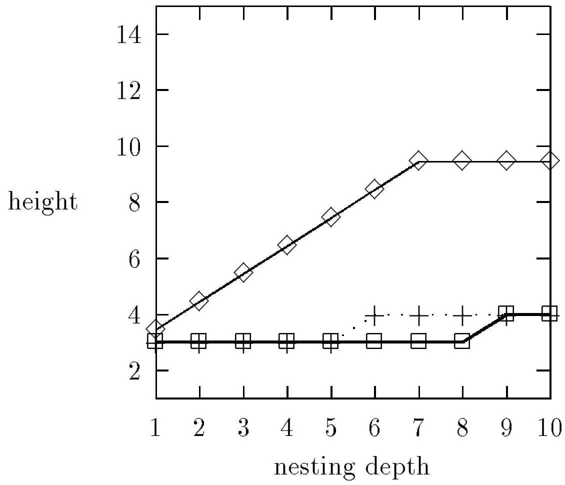


(b) uniform size

page size=1K



(c) non-uniform size



(d) uniform size

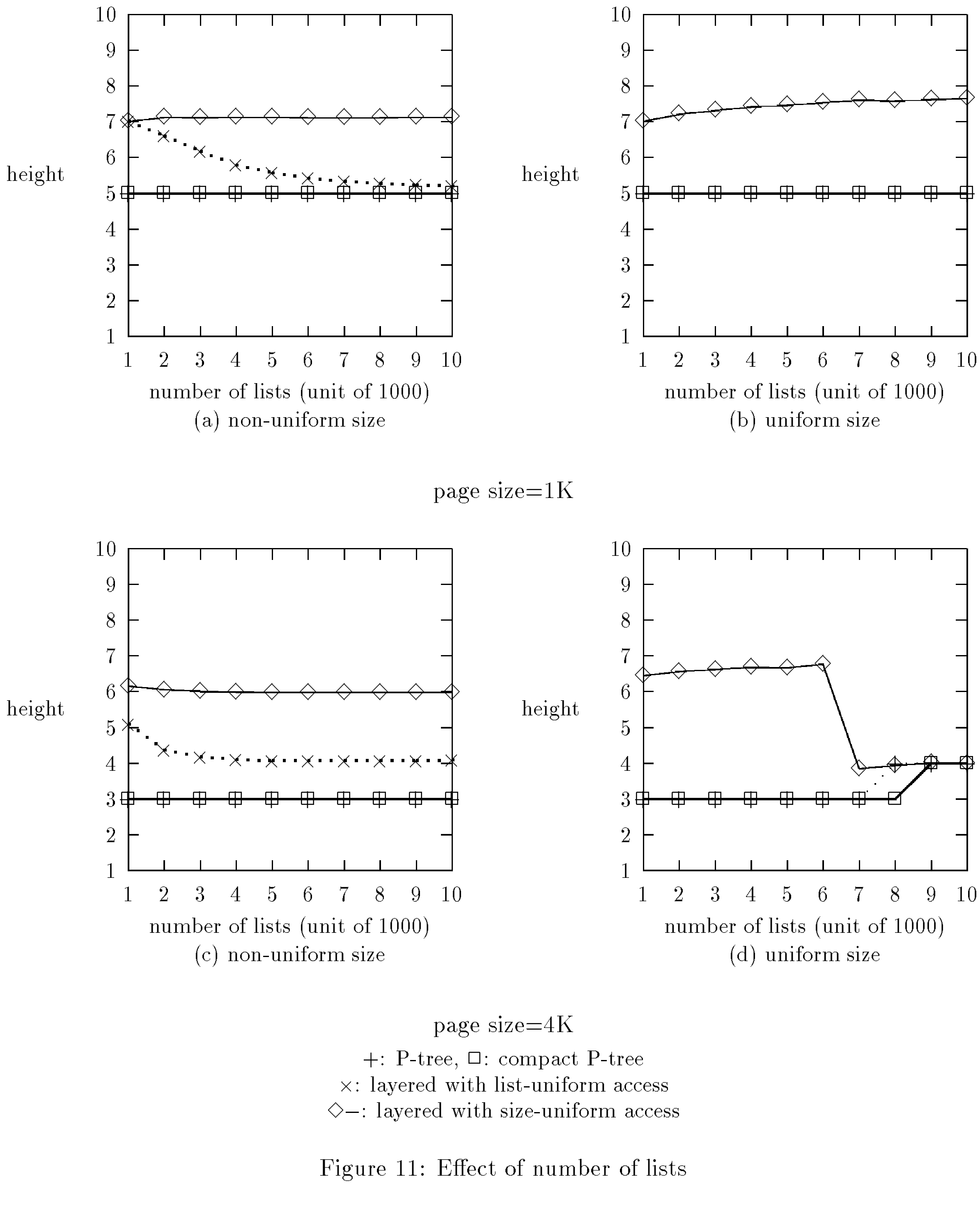
page size=4K

+: P-tree, □: compact P-tree

◇: layered with size-uniform access

×: layered with list-uniform access

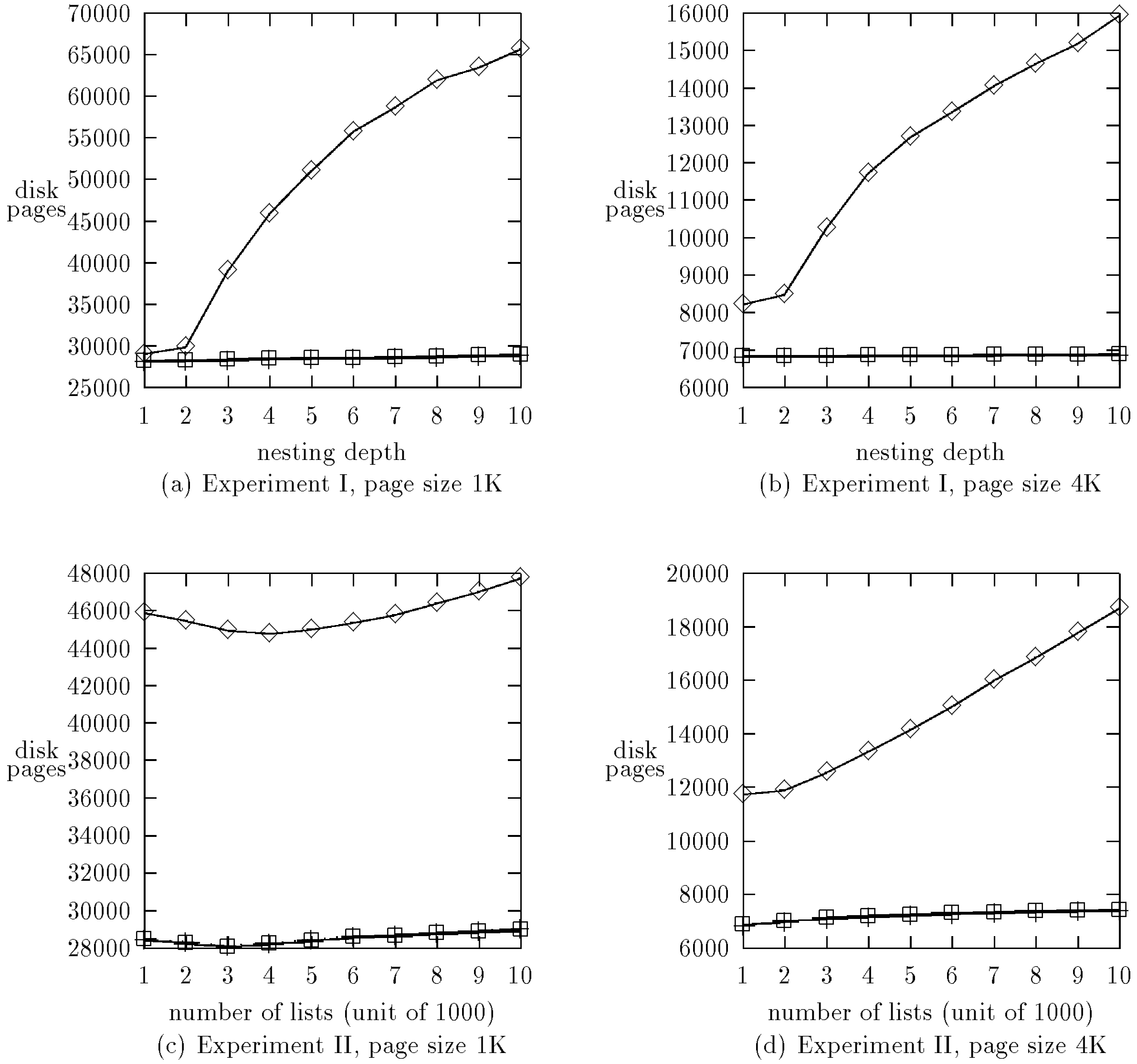
Figure 10: Effect of nesting depth



page size=4K

+: P-tree, □: compact P-tree
×: layered with list-uniform access
◇–: layered with size-uniform access

Figure 11: Effect of number of lists



+: P-tree, □: compact P-tree
◇: layered

Figure 12: Storage comparison