Adaptive Index Structures

Yufei Tao
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
taoyf@cs.ust.hk

Dimitris Papadias
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
dimitris@cs.ust.hk

Abstract

Traditional indexes aim at optimizing the node accesses during query processing, which, however, does not necessarily minimize the total cost due to the possibly large number of random accesses. In this paper, we propose a general framework for adaptive indexes that improve overall query cost. The performance gain is achieved by allowing index nodes to contain a variable number of disk pages. Update algorithms dynamically re-structure adaptive indexes depending on the data and query characteristics. Extensive experiments show that adaptive B- and R-trees significantly outperform their conventional counterparts, while incurring minimal update overhead.

1. Introduction

A single disk access usually includes (i) a *seek* operation that positions the disk head at the requested sector (including cylinder seek and rotation), and (ii) data transfer to/from the main memory. Thus the total query cost is the sum of disk seek, data transfer, and CPU time. Although significant advances have been made to accelerate CPU processing and data transfer, there is little progress on improving the seek time due to the mechanic nature of the disk head movement. The *average seek time* of latest models from major hard disk vendors, for example, is around 10ms (almost the same as 10 years ago), while the CPU costs and data transfer rates are usually 1-2 orders of magnitude smaller. Such performance difference is expected to become even greater in the future, which renders seek time the dominating factor in query cost.

Existing indexes focus on minimizing the number of node accesses required for query processing. Since a single node usually corresponds to a constant number (typically, one) of disk pages, fewer node accesses lead to a smaller number of page accesses. Minimizing the page accesses, however, does not necessarily optimize the total query cost. Consider, for example, that query q_1 accesses 20 pages whose addresses are consecutive, while q_2 must visit 10 random pages (i.e., non-consecutive ones). Then, the cost of q_1 is approximately $T_{SK}+20\cdot T_{TRF}+20\cdot T_{CPU}$, where T_{SK}, T_{TRF}, T_{CPU} are the costs of performing one disk seek (around 10ms), transferring one page (1ms/page), and processing the records in one page respectively (<0.1ms/page). Notice that only one disk seek is required to visit continuous pages (i.e., *sequential accesses*). Similarly, the cost of q_2 is $10\cdot T_{SK}+10\cdot T_{TRF}+10\cdot T_{CPU}$ (ten seeks must be performed to locate all pages, i.e., *random accesses*). Given that, T_{SK} is usually significantly more expensive (over an order of magnitude) than T_{TRF} and T_{CPU}, processing q_1 can be much cheaper than q_2.

The design of traditional indexes usually overlooks the difference between sequential and random accesses. Since the pages allocated to sibling nodes are often not consecutive, a query (such as q_2) may incur a large number of random accesses. The traditional method to reduce random accesses in databases (and general file systems) is to re-organize the data pages by de-fragmentation. This approach, however, has several serious drawbacks. First, re-organization involves some expensive operations: (i) moving (i.e., reading and writing) a large number of pages, and (ii) correcting mutual references (e.g., pointers from parent index nodes to their children, references to foreign keys, etc). Particularly, since references are ubiquitous (especially for databases with complex ER schemata), correcting them usually involves updating a very large part of the database. Second, a good page organization may soon degenerate by subsequent updates, in which case the benefit of re-organization vanishes in spite of its huge cost.

One approach to remedy this is to allocate several continuous pages to a node at a time. Thus, the query q_2 mentioned earlier visits one leaf node (with ten pages) to retrieve the same content, which reduces the random access (i.e., seek) time. Setting the node size to a fixed value, however, only favours queries with specific

Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002

selectivity; queries with different selectivity are optimized with different node sizes. Furthermore, to minimize the total cost, the data transfer and CPU time must be considered (in conjunction with the seek time).

Existing indexes are built by taking into account only the data distribution. In this paper, we introduce *adaptive index structures* that also consider query characteristics. Statistical information about query patterns is usually stored as histograms in the log of database systems. Taking advantage of this information to adapt indexes can improve their performance considerably. This improvement is achieved by using variable node sizes in different portions of the tree. In particular, each node size optimizes the average response time for the data and query distributions specific to the data space covered by the node. Carefully designed update algorithms allow adaptive indexes to re-structure as the distributions change. Analytical and experimental comparison proves that adaptive structures significantly outperform their conventional counterparts.[1]

The rest of the paper is organized as follows: section 2 introduces related work and section 3 describes the motivation and concrete algorithms of our adaptive framework using B-trees as an example. Section 4 extends the concept to R-trees and section 5 presents an extensive experimental evaluation to demonstrate the efficiency of the proposed methods. Section 6 concludes the paper with directions for future work.

2. Related Work

We focus on selection queries in traditional (i.e., B-trees) and multidimensional (R-trees) access methods. A selection query in relational databases specifies a 1D range [q_S, q_E] where $q_S \le q_E$, and retrieves all records *r* whose keys $r.key \in [q_S, q_E]$. Given a B-tree that indexes the keys, the query is processed as follows: first, the leaf page that contains the starting key q_S is located by following a path from the root to the leaf level; then, the sibling leaf pages are retrieved (by traversing the links among them) until the one that contains the ending key q_E has been reached. This is illustrated in Figure 2.1 (each page contains 3 entries), where the query [40, 75] visits nodes *H*, *F*, *C*, *D*, and *E*.

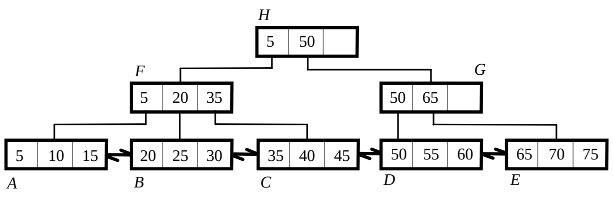
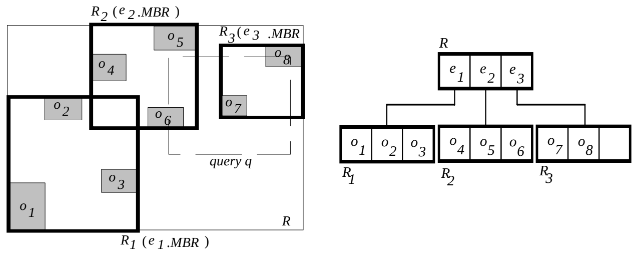


Figure 2.1: A conventional B-tree

The R-tree [G84, SRF87, BKSS90] can be viewed as a multi-dimensional extension of B-trees for spatial objects.

[1] Notice that the problem here is different from that of choosing a suitable page size for a file system, where a fixed size is decided when the system is set up.

Figure 2.2a shows an example where 7 objects (grey rectangles) are clustered into 3 leaf nodes R_1, R_2, and R_3 that are further grouped into 1 root node *R* (Figure 2.2b). Each non-leaf entry maintains a Minimum Bounding Rectangle (MBR) that encloses all objects in its sub-tree (e.g., e_1.*MBR* bounds all objects in R_1). The counterpart of selection queries in R-trees (and multi-dimensional access methods in general) is the *window query*, which specifies a rectangle *q* and retrieves all the objects that intersect *q*. The R-tree answers a window query *q* as follows. The root *R* is first retrieved and the entries inside it are compared with *q*. The child node of e_1 is not visited because its MBR does not intersect *q* (so none of the objects in its sub-tree can intersect *q*). On the other hand, non-leaf entries whose MBRs intersect *q* must be searched. As a result, in Figure 2.2, leaf nodes R_2 and R_3 are accessed, where objects o_6, o_7, and o_8 are retrieved.



(a) Grouping of rectangles (b) The R-tree

Figure 2.2: An R-tree example

The concept of variable node sizes has been applied to the X-tree [BKK96], an optimized version of R-trees for high-dimensional data. A *supernode* consists of numerous sequential pages in order to avoid node splitting that will lead to significant overlap between the MBRs of non-leaf entries. Whereas sizes of supernodes depend only on the data characteristics, node sizes of adaptive structures depend on both data and query distribution. Furthermore, the concept of supernode does not extend to other structures, while most tree indexes can be transformed to adaptive versions.

During the construction of an adaptive structure, we maintain a histogram that stores a small amount of statistical information about data and query distributions. The use of histograms is crucial for effective query optimization, and has received considerable research attention. Existing approaches can be classified into two categories depending on whether they take into account only the data distribution [HS92, IP95, GM98, APR99, WAA01], or also consider the query patterns [CR94, GLR00, BCG01, WAA02]. Although our framework can be used with any histogram, for the shake of simplicity and generality, we adopt the “equi-length” method (in fact more sophisticated histograms lead to even better performance). Specifically, the data space is divided into num_{bin} *bins* with equal extents, and statistical information is maintained for each bin$_i$ ($1 \le i \le num_{bin}$) individually. Obviously larger num_{bin} leads to better estimation, but increases the storage and computational overhead.

3. Adaptive B-Trees

The general idea of adaptive B-trees is to allow nodes to span several pages based on the average query length in the corresponding part of the data space. Figure 3.1 shows an adaptive version of the B-tree in Figure 2.1 assuming that the expected selectivity of queries in the range [5,30] corresponds to two pages, while that of queries in the range [35,75] corresponds to three pages. The sizes of nodes *A* and *B* are 2 and 3 respectively, following the expected selectivity. The disk pages assigned to the same node are consecutive so that, after locating the first page, accesses to the others are sequential. All non-leaf nodes have the minimum size (one page per node). Notice that, the height of the tree is lower than in Figure 2.1 because the number of leaf nodes is smaller. A query [40, 75] only needs to visit 2 nodes (a total of 4 pages) incurring 2 random accesses, instead of 5 for the B-tree in Figure 2.1.

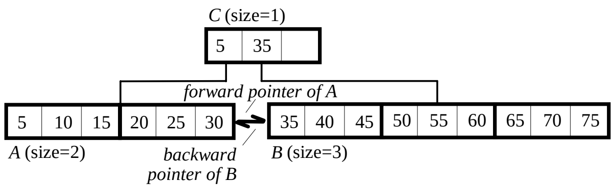


Figure 3.1: An adaptive B-tree example

Adaptive B-trees, like their traditional versions, ensure minimum node utilization (typically, 50%). However, since nodes have variable capacity, the minimum number of entries in a node may vary. The capacities of nodes *A*, *B* in Figure 3.1, for example, are 6 and 9 entries respectively; as a result their minimum utilization is 3 and 5. An overflow (underflow) occurs when the number of entries in a node is beyond (below) the capacity (the utilization).

Section 3.1 derives the optimal node size based on a model that predicts performance in terms of node accesses, and sections 3.2 and 3.3 describe update algorithms. The performance of adaptive B-trees is analyzed in section 3.4, while section 3.5 discusses bulkloading and partial rebuilding techniques.

3.1 Optimal node size

Each node in the B-tree is associated with a range of keys, which we call the *extent* of the node. The extent of node *A* in Figure 2.1, for example, is [5, 20). A node will be visited by a range query [q_S, q_E] if and only if its extent intersects the query range. In particular, for uniform data distribution, extents of nodes at the same level are approximately the same. Assuming that *f* is the average fanout of a node, and *N* the total number of records indexed, the extent of a leaf node corresponds to *f* / *N* of the entire data space (assumed to be a unit line segment in the sequel). The probability PR_{INTS} that the extent of a node intersects a query with length q_L=$q._S$–$q._E$ is [TSS00]:

$$PR_{INTS} = q_L + f/N$$

Since the total number of leaf nodes is *N* / *f*, the expected number of leaf node accesses is:

$$NA_{LEAF}(q_L) = N/f \cdot PR_{INTS} = N/f \cdot q_L + 1$$

If each node contains *p* pages, *f* equals $\xi \cdot p \cdot b_{sp}$ where b_{sp} is the maximum number of entries in a single page (i.e., $p \cdot b_{sp}$ corresponds to the node capacity), and ξ the average node utilization (common value 69% [TSS00]). A node access involves the following costs: (i) T_{SK} time to perform a disk seek operation, (ii) $p \cdot T_{TRF}$ time to transfer *p* pages to the memory, and (iii) $f \cdot T_{EVL}$ (CPU) time to process all the entries in the node. Hence, the overall query time at the leaf level is:

$$TIME_{LEAF}(q_L) = NA_{LEAF}(q_L) \cdot (T_{SK} + p \cdot T_{TRF} + f \cdot T_{EVL}) \quad (3.1\text{-}1)$$
$$= \left(N/f \cdot q_L + 1 \right) \cdot (T_{SK} + p \cdot T_{TRF} + f \cdot T_{EVL})$$

Taking the derivative of the above equation with respect to *p* we have (after applying $f=\xi \cdot p \cdot b_{sp}$):

$$d\frac{TIME_{LEAF}(q_L)}{dp} = T_{TRF} + \xi \cdot b_{sp} \cdot T_{EVL} - \frac{q_L \cdot N \cdot T_{SK}}{\xi \cdot b_{sp}} \cdot \frac{1}{p^2}$$

The optimal node size that minimizes $TIME_{LEAF}$ can be obtained by setting the above equation to 0:

$$p_{OPT}(q_L) = \left\lceil \sqrt{\frac{q_L \cdot N \cdot T_{SK}}{\xi \cdot b_{sp}\left(T_{TRF} + \xi \cdot b_{sp} \cdot T_{EVL}\right)}} \right\rceil \quad (3.1\text{-}2)$$

This equation suggests that the optimal node size depends on both data and query characteristics. Specifically, higher *N* or longer q_L leads to larger optimal size as the number of retrieved records increases (thus, more data pages are retrieved). A faster CPU or higher data transfer rate (i.e., smaller T_{EVL} or T_{TRF} respectively) also increases the size because the I/O cost will account for a higher percentage in the total cost. The optimal size given in (3.1-2) is for leaf nodes, while for non-leaf nodes, the optimal size is always 1, because only one node is accessed at each non-leaf level. Setting the node size higher results in the same seek time (of 1 disk seek) yet higher data transfer and evaluation cost.

The discussion so far assumes the same range length q_L for all queries, whereas in practice queries have arbitrary length/selectivity. Assume an array q_L[] = {q_L[1], q_L[2], …, q_L[*t*]} that stores all lengths (*t* different values) of queries recorded so far, and an array *pr*[], where *pr*[*i*] is the probability for length q_L[*i*] to appear in a query. The expected processing time (at the leaf level) of a query *q* that conforms to this distribution is:

$$Exp(TIME_{LEAF}) = \sum_{i=1}^{t} pr[i] \cdot TIME_{LEAF}(q_L[i]) \quad (3.1\text{-}3)$$
$$= \sum_{i=1}^{t} pr[i] \cdot \left(N/f \cdot q_L[i] + 1 \right) \cdot (T_{SK} + p \cdot T_{TRF} + f \cdot T_{EVL})$$
$$= \left(N/f \sum_{i=1}^{t} pr[i] \cdot q_L[i] + \sum_{i=1}^{t} pr[i] \right) (T_{SK} + p \cdot T_{TRF} + f \cdot T_{EVL})$$
$$= \left(N/f \cdot Exp(q_L) + 1 \right) (T_{SK} + p \cdot T_{TRF} + f \cdot T_{EVL})$$

The above equation, when compared with (3.1-1), indicates an interesting fact: *optimizing a query distribution is equivalent to optimizing a single query* *whose length equals the expected range length* ($Exp(q_L)$). Thus, the optimal node size is also given by equation (3.1-2), except that q_L is replaced with $Exp(q_L)$. Furthermore, since $Exp(q_L)$ is the only information required, the amount of statistics that must be kept can be significantly reduced. To verify this, observe that $pr[i]$ can be written as $num\_q[i]/total\_q$, where $num\_q[i]$ is the number of times that queries with range length $q_L[i]$ are raised, and $total\_q$ is the total number of queries. Therefore, $Exp(q_L)$ can be represented as:

$$Exp(q_L) = \sum_{i=1}^{t}\left(q_L[i]\cdot\frac{num\_q[i]}{total\_q}\right)$$
$$= \frac{1}{total\_q}\cdot\sum_{i=1}^{t}\left(q_L[i]\cdot num\_q[i]\right) = \frac{sum\_q_L}{total\_q}$$

where sum_q_L is the sum of the range lengths of all queries. Therefore, instead of maintaining $q_L[]$ and $pr[]$, it suffices to keep values of sum_q_L and $total\_q$.

In order to extend the above approach from uniform to arbitrary distributions, we maintain an "equi-length" histogram with the following information for each bin$_i$ ($1\le i\le num_{bin}$): (i) the number (n_i) of records in the bin (ii) the range sum (sum_q_{Li}) of queries that intersect the extent of bin$_i$ (for a query that does not lie completely in the bin, only the covered part is added to sum_q_{Li}), and (iii) the number $total\_q_i$ of queries in bin$_i$. The node accesses for performing a query with range q_L in bin$_i$ are estimated as [TPZ01]:

$$NA_{LEAF}(q_L) = {n_i\cdot num_{bin}\cdot q_L}\big/{f} + 1 \qquad (3.1\text{-}4)$$

Using this estimation, we obtain the optimal node size P_{OPTi} for bin$_i$ (nodes in the same bin have the same size) that minimizes the expected cost of queries in bin$_i$ as:

$$p_{OPTi} = \left\lceil \sqrt{\frac{Exp(q_{Li})\cdot n_i\cdot num_{bin}\cdot T_{SK}}{\xi\cdot b_{sp}\left(T_{TRF}+\xi\cdot b_{sp}\cdot T_{EVL}\right)}}\right\rceil \qquad (3.1\text{-}5)$$

where $Exp(q_{Li})=sum\_q_{Li}/total\_q_i$. In the following sections we elaborate the update algorithms for adaptive B-trees. Table 3.1 summarizes the symbols that will be used frequently.

T_{SK}	average seek time of a random access
T_{TRF}	time for transferring one disk page to/from the main memory
T_{EVL}	CPU time to evaluate a single record
b_{sp}	number of entries contained in a page
ξ	average node utilization
f	average fanout of a node
h	height of an index
N	total number of records
q_L	range length of a query
num_{bin}	number of bins in the histogram
n_i	number of records in bin$_i$
sum_q_{Li}	sum of query range lengths in bin$_i$
$total\_q_i$	number of queries that intersect *bin$_i$*

Table 3.1: Summary of frequent symbols

3.2 Insertion and overflow handling

Since an adaptive B-tree is maintained with the aid of a histogram, both the tree and histogram must be updated in an insertion. Updating the histogram, however, incurs little cost: it suffices to increment the number n_i of records in bin$_i$ where the new record is inserted. Furthermore, since the size of a histogram is very small (usually less than 1K bytes), this operation can be performed in memory. An insertion in the adaptive B-tree, on the other hand, is performed in a way similar to B-trees. The leaf node that accommodates the new entry is identified by examining the index keys of non-leaf nodes at each level (we omit the details since the process is the same as conventional B-trees), and the insertion terminates if no overflow occurs.

When a node P generates an overflow, its optimal size is re-computed using equation (3.1-5), based on the *current* information (i.e., n_i, sum_q_{Li}, and $total\_q_i$) stored in the bin$_i$ ($1\le i\le num_{bin}$) that contains the extent of P. If the extent of node P intersects multiple bins, information is obtained from the bin that covers the largest part of P. Note that, because statistical information may have changed considerably since the last time that the size of P was computed (due to significant changes in the query patterns), the new optimal size can be much larger or smaller than its original value. In the sequel, we denote the old and new sizes of P as $P.\text{Size}_{\text{old}}$ and $P.\text{Size}_{\text{new}}$ respectively. In adaptive B-trees, an overflow is handled by examining the relationship between $P.\text{Size}_{\text{old}}$ and $P.\text{Size}_{\text{new}}$, as summarized in Figure 3.2. Next, we discuss each case in detail.

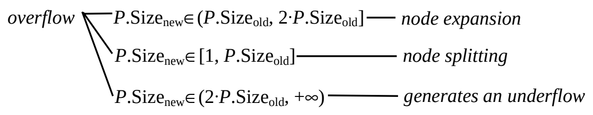

overflow
$P.\text{Size}_{\text{new}}\in(P.\text{Size}_{\text{old}},\ 2\cdot P.\text{Size}_{\text{old}}]$
node expansion
$P.\text{Size}_{\text{new}}\in[1,\ P.\text{Size}_{\text{old}}]$
node splitting
$P.\text{Size}_{\text{new}}\in(2\cdot P.\text{Size}_{\text{old}},\ +\infty)$
generates an underflow

Figure 3.2: Overflow handling in adaptive B-trees

If $P.\text{Size}_{\text{new}}\in(P.\text{Size}_{\text{old}},\ 2\cdot P.\text{Size}_{\text{old}}]$, no overflow is handled; we only need to expand node P to its new size, after which the number (i.e., $b_{sp}\cdot P.\text{Size}_{\text{old}}+1$) of entries in P is within the range of $[½\cdot b_{sp}\cdot P.\text{Size}_{\text{new}},\ b_{sp}\cdot P.\text{Size}_{\text{new}}]$ (i.e., greater than the minimum node utilization yet smaller than the node capacity). Since we aim at allocating sequential pages to a node, special care must be taken during the node expansion, if there are not enough consecutive vacant pages after those originally allocated to P. This is illustrated in Figure 3.3, where white blocks indicate vacant disk pages, grey blocks the occupied pages, and black blocks (page IDs 4, 5) the pages originally assigned to P, whose size must be expanded from 2 (pages) to 4. Since there is only one vacant page (i.e., block 6) after page 5, the content of P must be migrated to another place with at least 4 consecutive empty pages (e.g., blocks 9-12 in this case). Its original pages are marked vacant (freed) for future use.

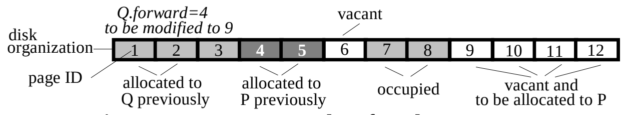


Figure 3.3: An example of node expansion

Because in adaptive B-trees the leaf pages are organized as a linked-list, moving the starting address of *P* requires updating the corresponding reference in its previous sibling node (denoted as *Q*). In Figure 3.3, the forward pointer of *Q*, which originally pointed to block 4, must be updated to 9 after the node expansion. Note that *Q* can be located via the "backward" pointer stored in *P* without traversing the non-leaf levels. The parent entry that points to *P* must also be updated. Figure 3.4 presents the pseudo-code for node expansion.

```
Algorithm **Node_Expansion** (P: the node that overflows)
1.  if there are at least $P.\text{Size}_{\text{new}}-P.\text{Size}_{\text{old}}$ vacant pages subsequent to currently assigned pages of P
2.      mark these pages occupied and return
3.  else /*no enough vacant pages and migration is necessary*/
4.      allocate $P.\text{Size}_{\text{new}}$ consecutive pages and copy the content of P to the newly allocated pages
5.      free the original pages of P (i.e., mark them vacant)
6.      retrieve the previous sibling Q of P via P.backward, and set Q.forward=P
7.      modify the reference in the parent entry of P
end **Node_Expansion**
```

Figure 3.4: Algorithm for node expansion

In lines 1 and 2 we need to check/modify the vacancy status of a set of disk pages, which can be done by simply maintaining a bitmap where each bit corresponds to the status of one page. Such an approach is widely adopted in operating systems. The size of the bitmap is usually small enough to fit in the memory. For example, for an index with 1G bytes and formatted into 1K bytes/page, the bitmap contains 1M bits, which amounts to less than 130K memory.

For the second case ($P.\text{Size}_{\text{new}} \le P.\text{Size}_{\text{old}}$) an overflowing node *P* is split into several (≥2) nodes by distributing the entries evenly. There are multiple ways to decide the number NM_{SPLT} of resulting nodes so that the number of entries in each node is within the range [$½\cdot P.\text{Size}_{\text{new}}$, $P.\text{Size}_{\text{new}}$]. We use equation (3.2-1), which computes the minimum among all legal values for NM_{SPLT}. Minimizing the number of nodes after splitting also reduces the number of nodes accessed during a query.

$$NM_{SPLT} = \left\lceil \left(b_{sp} \cdot P.Size_{old} + 1\right) \Big/ \left(b_{sp} \cdot P.Size_{new}\right) \right\rceil \qquad (3.2\text{-}1)$$

Assume, for example, we are to insert a new key 80 into the tree of Figure 3.1, which generates an overflow in node *B*. The original size of *B* was 3, while, due to the change of histogram information, its new optimal size is 2. Then, NM_{SPLT} is computed as $\lceil(3\cdot3+1)/(3\cdot2)\rceil=2$, and the original entries are evenly distributed into two nodes *B* and *D*. A parent entry is inserted into the root *C*, and the final situation is shown in Figure 3.5. In general, overflow may propagate to nodes at higher levels, which are handled in the same way as in conventional B-trees (since their size is always 1).

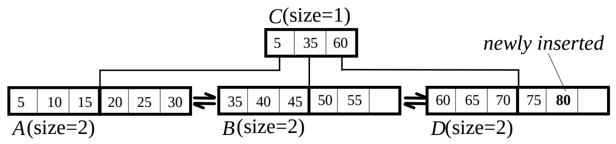


Figure 3.5: A node splitting example

Like node expansion, splitting may cause a set of consecutive disk pages to re-allocate. In some rare scenarios (e.g., the disk is over-fragmented) when such allocation fails, we gradually decrease $P.\text{Size}_{\text{new}}$ until such allocation is possible, which is always the case when $P.\text{Size}_{\text{new}}$ reaches the minimum size 1 (i.e., at this point the adaptive B-tree allocates pages in the same way as conventional B-trees).

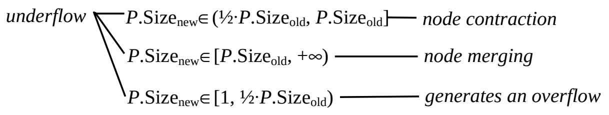
```
Algorithm **Leaf_Split** (P: the node that needs to be split)
1.  calculate $NM_{SPLT}$ as in equation (3.2-1)
2.  distribute entries in P evenly into $NM_{SPLT}$ nodes
end **Leaf_Split**

Algorithm **Insert** (new_e: the entry to be inserted, P: the current node being processed)
1.  if P is a leaf node
2.      enter new_e in P
3.      if P does not overflow
4.          return
5.      calculate the new size $P.\text{Size}_{\text{new}}$ for P
6.      if $P.\text{Size}_{\text{new}} \in (P.\text{Size}_{\text{old}}, 2\cdot P.\text{Size}_{\text{old}}]$
7.          call **Node_Expansion**(P) and return
8.      elseif $P.\text{Size}_{\text{new}} \le P.\text{Size}_{\text{old}}$
9.          call **Leaf_Split**(P) and return
10.     else /*$2\cdot P.\text{Size}_{\text{old}} < P.\text{Size}_{\text{new}}$*/
11.         call **Leaf_Merge**(P) and return /*Leaf_Merge is presented in the next section*/
12. else /*a non-leaf node*/
13.     find the child node c_P to insert new_e /*this process is same as B-trees*/
14.     call **Insert**(new_e, c_P)
15.     add/remove/modify pointers to child nodes (for node split, expansion, or merging)
16.     if P overflows that split into 2 nodes with even entries /*as in ordinary B-trees*/
end **Insert**
```

Figure 3.6: Algorithm for leaf splitting and insertion

In the last case where $P.\text{Size}_{\text{new}} > 2\cdot P.\text{Size}_{\text{old}}$, an underflow is generated because the new optimal size has increased significantly so that the number of entries in the original node is not enough for maintaining the minimum node utilization in the new node. This underflow is handled by node merging as described in the next section. Figure 3.6 summarizes the leaf split and insertion procedures.

3.3 Deletion and underflow handling

Deletion of an entry *e* is performed by (i) first locating the leaf node *P* that contains *e*, (ii) removing *e* from *P*, and (iii) handling the node underflow if the number of entries in *P* is below the minimum node usage. Steps (i) and (ii) are the same as in B-trees, so we elaborate (iii) in the sequel. Similar to overflows, the size of a leaf node P is re-computed when it underflows, and the handling proceeds by examining the relationship between $P.\text{Size}_{\text{new}}$ and $P.\text{Size}_{\text{old}}$, as shown in Figure 3.7.



underflow
$P.\text{Size}_{\text{new}} \in (½\cdot P.\text{Size}_{\text{old}}, P.\text{Size}_{\text{old}}]$ — *node contraction*
$P.\text{Size}_{\text{new}} \in [P.\text{Size}_{\text{old}}, +\infty)$ — *node merging*
$P.\text{Size}_{\text{new}} \in [1, ½\cdot P.\text{Size}_{\text{old}})$ — *generates an overflow*

Figure 3.7: Underflow handling in adaptive B-trees

Node contraction is performed if $P.\text{Size}_{\text{new}} \in [½\cdot P.\text{Size}_{\text{old}}, P.\text{Size}_{\text{old}})$. Specifically, a node contraction simply reduces the size of a node to its new value, by freeing the "tailing pages" originally assigned to P. Assume, for example, the size of node P in Figure 3.3 is reduced from 2 (pages) to 1; then block 5 becomes vacant for future use. Notice that, unlike node expansion, no content migration is necessary.

If $P.\text{Size}_{\text{new}} \geq P.\text{Size}_{\text{old}}$, P is merged with one or more sibling nodes. To illustrate this, we successively remove entries 45, 50, 55 from node B in Figure 3.5, causing it to underflow. Assume its new optimal node size is 5, (capacity is 15 entries). Then, a sibling (let D) is chosen to merge with B, which leads to a total of 7 entries (2, and 5 from B, D respectively) in the resulting node. Since 7 is still smaller than the minimum node utilization (i.e., $\lceil 15/2 \rceil$=8 entries), the merging is continued with another sibling (node A in this case), after which the final node contains 13 entries and the merging step terminates (in general the merged node may need to be split if it overflows). This process is shown in Figure 3.8, where the deletion propagates to upper levels, which results in only 1 level in the final tree. Figure 3.9 presents the pseudo-code for merging leaf nodes; merging non-leaf nodes is identical to B-trees.

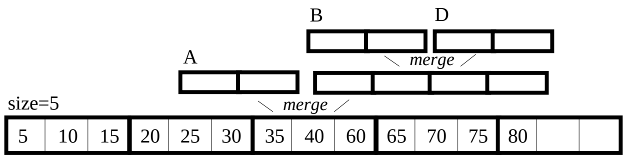


Figure 3.8: A node merging example

```
Algorithm Leaf_Merge (P: the current leaf node)
1.  calculate the new size $P.\text{Size}_{\text{new}}$ for P, and let P.num be the
    number of entries in P
2.  allocate $P.\text{Size}_{\text{new}}$ continuous disk pages and copy the
    content of P to these pages
3.  free the original pages of P
4.  while ($P.\text{num} < ½\cdot P.\text{Size}_{\text{new}}$)
5.      identify a sibling node S
6.      copy the content of S to P, and free the pages of S
7.      P.num=P.num+S.num
8.      if $P.\text{num} > P.\text{Size}_{\text{new}}$ then call Node_Split(P) /*the
        merged node overflows*/
end Insert
```

Figure 3.9: Algorithm for node merging

In the third case ($P.\text{Size}_{\text{new}} < ½\cdot P.\text{Size}_{\text{old}}$), an overflow is generated, which is handled by node splitting. Note that this is the reverse of the case $P.\text{Size}_{\text{new}} > 2\cdot P.\text{Size}_{\text{old}}$ in node overflows. The entire deletion algorithm is presented in Figure 3.10.

```
Algorithm Delete (new_e: the entry to be deleted, P: the current
                  node being processed)
1.  if P is a leaf node
2.      remove new_e from P
3.      if P does not underflow then return
4.      calculate the new size $P.\text{Size}_{\text{new}}$ for P
5.      if $P.\text{Size}_{\text{old}} \in (P.\text{Size}_{\text{new}}, 2\cdot P.\text{Size}_{\text{new}}]$
6.          call Node_Contraction(P) and return /*we omit
            Node_Contraction because it is straightforward*/
7.      elseif $P.\text{Size}_{\text{old}} \leq P.\text{Size}_{\text{new}}$
8.          call Leaf_Merge(P) and return
9.      else /*$P.\text{Size}_{\text{old}} > 2\cdot P.\text{Size}_{\text{new}}$ */
10.         call Leaf_Split(P) and return /*Leaf_Split was
            presented in the last section*/
11. else /*a non-leaf node*/
12.     find the child node c_P to find new_e /*this process is
        same as B-trees*/
13.     call Delete(new_e, c_P)
14.     add/remove/modify pointers to child nodes (for node
        split, expansion, or merging)
15.     if P underflows that handles it as in ordinary B-trees
end Delete
```

Figure 3.10: Algorithm for deletion

3.4 Performance of adaptive B-trees

We first show that the space complexity of adaptive B-trees is $O(N/b_{ps})$ pages (i.e., asymptotically optimal), where N is the total number of records indexed, and b_{ps} the maximum number of entries in a single page. Let n_{leaf} be the total number of leaf nodes, and ns_i ($1 \leq i \leq n_{leaf}$) the size of the ith node. Since each node is at least half full, we have:

$$\sum_{i=1}^{n_{leaf}} \frac{1}{2}\cdot ns_i \cdot b_{ps} \leq N \Rightarrow \sum_{i=1}^{n_{leaf}} ns_i \leq 2N/b_{ps} = O\left(N/b_{ps}\right)$$

Furthermore, because each non-leaf node indexes at least $½\cdot b_{ps}$= $O(b_{ps})$ children, the total number of nodes at level j ($1 \leq j \leq h$-1, where h is the height) is $O(N/b_{ps}{}^{j+1})$, which establishes the optimal size bound:

$$Size_{QAB} = \sum_{i=0}^{h-1} O\left(N/b_{ps}{}^{i+1}\right) < 2O\left(N/b_{ps}\right) = O\left(N/b_{ps}\right)$$

In fact, the size of an adaptive B-tree is usually slightly smaller than that of its conventional counterpart because it contains fewer intermediate nodes. Similarly, it can be shown that the performance of adaptive B-trees for range queries is also asymptotically optimal: $O(\log_b N + K/b_{sp})$, where K is the number of records retrieved.

Adapting the cost model of B-trees to adaptive B-trees is straightforward. Consider a query q in bin_i ($1 \leq i \leq bin_{num}$). Since nodes in bin_i have the same size P_i, by replacing f in equation (3.1-4) with $\xi\cdot b_{sp}\cdot P_i$, we have (q_L is the range length of the query):

$$NA_{LEAF}(q_L) = \frac{n_i \cdot num_{bin} \cdot q_L}{\xi \cdot b_{sp} \cdot P_i} + 1$$

Given that one node (with size 1) is accessed at each non-leaf level, the total query response time is:

$$TIME_{QAB}\left(q_L\right) = \left(h-1\right)\cdot\left(T_{SK}+T_{TRF}+\xi\cdot b_{spi}\cdot T_{EVL}\right) + \left(\frac{n_i\cdot num_{bin}\cdot q_L}{\xi\cdot b_{sp}\cdot P_i}+1\right)\left(T_{SK}+P_i\cdot T_{TRF}+\xi\cdot b_{sp}\cdot P_i\cdot T_{EVL}\right) \quad (3.3\text{-}1)$$

The performance speedup over conventional B-trees can be obtained by comparing the above cost model with that of B-trees:

$$Speedup = \frac{TIME_B\left(q_L\right)}{TIME_{QAB}\left(q_L\right)} = \frac{\left[\begin{array}{l}\left(h_B-1\right)\left(T_{SK}+T_{TRF}+\xi\cdot b_{sp}\cdot T_{EVL}\right)\\ +\left(\frac{n_i\cdot num_{bin}\cdot q_L}{\xi\cdot b_{sp}}+1\right)\left(T_{SK}+T_{TRF}+\xi\cdot b_{sp}\cdot T_{EVL}\right)\end{array}\right]}{\left[\begin{array}{l}\left(h_{QAB}-1\right)\left(T_{SK}+T_{TRF}+\xi\cdot b_{sp}\cdot T_{EVL}\right)\\ +\left(\frac{n_i\cdot num_{bin}\cdot q_L}{\xi\cdot b_{sp}\cdot P_i}+1\right)\left(T_{SK}+P_i\cdot T_{TRF}+\xi\cdot b_{sp}\cdot P_i\cdot T_{EVL}\right)\end{array}\right]} \quad (3.3\text{-}2)$$

For large n_i or q_L (i.e., high data cardinality or long query ranges) the speedup converges to:

$$Speedup \rightarrow \frac{T_{SK}+T_{TRF}+\xi\cdot b_{sp}\cdot T_{EVL}}{T_{TRF}+\xi\cdot b_{sp}\cdot T_{EVL}} \quad (3.3\text{-}3)$$

For shorter queries, however, the speedup diminishes as q_L decreases. Obviously, if all queries are equality selections (q_L =0), the adaptive tree degenerates to a conventional B-tree.

The update cost is also very closely related to the node size. If the size of the leaf node P to be updated is P.Size, then the update needs to visit O(log$_b N$+P.Size) disk pages in the worst case. In order to achieve the optimal update cost bound, we may prevent the node from growing beyond a constant size by setting an upper limit C_{SIZE} to the node size, so that O(log$_b N$+P.Size) = O(log$_b N$+C_{SIZE}) = O(log$_b N$). To achieve this, the update algorithms need to set the node size to the minimum of the decided value and C_{SIZE}. Notice that the resulting tree still maintains the asymptotically optimal query performance.

In practice, if minimizing the update time is important (e.g., for frequent updates), the (leaf) node size computation can consider both the update and query frequencies. Specifically, we may maintain a separate histogram (also very small in size) that stores, for each bin$_i$, the update frequency u_i ($0\le u_i\le 1$) among all updates and queries in the bin (so the query frequency can be obtained as $1-u_i$). Since an update usually reads and writes a leaf page once, its cost (at the leaf level) can be represented as:

$$TIME_{LEAFUpdt} = 2\left(T_{SK}+P_i\cdot T_{TRF}+\xi\cdot b_{sp}\cdot P_i\cdot T_{EVL}\right)$$

Thus the expected weighted cost of updates and queries is:

$$TIME_{LEAFU+Q} = TIME_{LEAFUpdt}\cdot u_i + TIME_{LEAF}\cdot(1-u_i)$$

where $TIME_{LEAF}$ is given in equation (3.1-3). Therefore, the node size that minimizes the above cost can be obtained as:

$$d\,\frac{TIME_{LEAFU+Q}}{dp} = 0 \Rightarrow \quad (u_i<1)$$

$$p_{OPTi} = \left\lceil\sqrt{\frac{(1-u_i)\cdot Exp(q_{Li})\cdot n_i\cdot num_{bin}\cdot T_{SK}}{(1+u_i)\cdot\xi\cdot b_{sp}\cdot\left(T_{TRF}+\xi\cdot b_{sp}\cdot T_{EVL}\right)}}\right\rceil$$

For u_i=1 (i.e., only updates), P_{OPTi}=1.

3.5 Bulkloading and partial rebuilding

If all the data are known a-priori, the adaptive B-tree can be bulkloaded efficiently from the list of records sorted by their index keys. The algorithm differs from traditional bulkloading in that, whenever a leaf node is initiated, its size is determined according to the information stored in the histogram bin that covers the starting key of the node. A new node is initiated after the previous one has been fully filled.

Although bulkloading allocates continuous pages to sibling nodes, the organization of the conventional B-tree will deteriorate due to subsequent updates. This is illustrated in Figure 3.11, where pages 1-100 correspond to the leaf nodes of a bulk-loaded conventional B-tree (i.e., adjacent pages correspond to sibling nodes). Assume, for example later changes cause page 2 to split, which results in a new node stored in page 201, breaking the adjacency between pages 2 and 3. Thus, a query that retrieves these pages must also visit page 201 (hence accesses to pages 2, 201, 3 are no longer sequential). Similarly, the split of page 3 (leading to page 202) breaks the adjacency of pages 3 and 4.

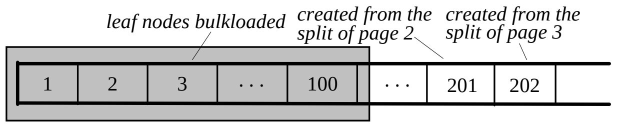

leaf nodes bulkloaded
created from the split of page 2
created from the split of page 3
1
2
3
. . .
100
. . .
201
202

Figure 3.11: Deterioration of traditional bulkloading

In general, each node split will necessarily break the adjacency of the bulk-loaded pages. Thus, the benefit of bulkloading vanishes completely when all pages 1-100 issue splits (after which all node visits are random accesses). To see how "soon" this can happen, assume that a node contains on average f entries after bulkloading (i.e., N/f leaf nodes); then a split incurs after b_{sp}–f entries are inserted in the node. It follows that all nodes may split after $N/f\cdot\left(b_{sp}-f\right)$ insertions. On the other hand, the update algorithms described in previous sections allow the adaptive B-tree to restructure, by re-computing the size of a node each time it generates an over-/under- flow.

A problem occurs when the data are relatively static (i.e., there are no overflows or underflows to trigger re-organization) whereas the query patterns change significantly. In this case an alternative mechanism, called *partial rebuilding*, restructures the bins whose optimal node size has changed. To *partially* rebuild the nodes in a single bin efficiently, we adopt an approach that is similar to the bulkloading algorithm. As shown in Figure 3.12, the process starts from the node A_1 whose extent is the leftmost in the bin, and copies its content to a new node B_1 whose size is optimized. If the entries in A_1 are exhausted, rebuilding proceeds with its sibling node A_2 until all nodes in the bin have been processed (nodes X and Y are not modified as they do not belong to the bin). Similarly, if node B_1 has been filled, a new node B_2 is initiated, and this process is repeated. Nodes in higher levels are constructed similarly. After rebuilding, the forward (backward) pointer of node X (Y) is modified to B_1 (B_v), and all disk pages assigned to the original nodes (e.g., A_1, …, A_u) are freed for future use.

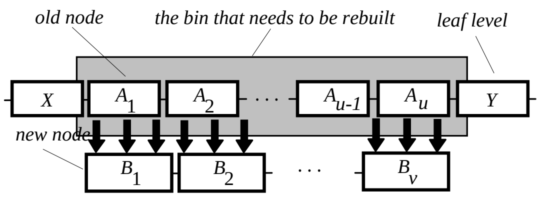


Figure 3.12: Partial rebuilding

Let p_{old} and p_{new} denote the old and new node sizes respectively; then rebuilding the leaf level involves reading $n_i/(\xi \cdot b_{sp} \cdot p_{old})$ and writing $n_i/(\xi \cdot b_{sp} \cdot p_{new})$ nodes. Since the number of intermediate nodes that need to be modified is significantly smaller than that of leaf nodes, the total rebuilding cost is dominated by the time of processing the leaf level:

$$Time_{rebuild} \approx \frac{n_i}{\xi \cdot b_{sp} \cdot p_{old}}(T_{SK} + p_{old} \cdot T_{TRF}) + \frac{n_i}{\xi \cdot b_{sp} \cdot p_{new}}(T_{SK} + p_{new} \cdot T_{TRF})$$

Note that nodes (e.g., A_1, …, A_u in Figure 3.12) can still be used to answer queries during rebuilding since their pages are freed only after the process terminates, at which point the new nodes (e.g., B_1, …, B_v) are integrated into the tree. Moreover, rebuilding of a bin is necessary only when (i) very few (or zero) nodes in the bin incur structural changes (otherwise restructuring is performed by overflow / underflow handling), and (ii) the expected q_L in the bin has deviated from its previous value over a certain threshold (otherwise the performance drop is not significant). In fact, since the query patterns in many applications are relatively stable [WAA02] (especially for patterns recorded over long periods), we believe that rebuilding is rare in practice.

4. Adaptive R-Trees

The method of converting B-trees to adaptive B-trees can be extended to general structures using analytical models for the number of node accesses. Specifically, the framework involves two steps: (i) deciding the optimal node size as a function of data and query parameters, and (ii) modifying the original update algorithms with the following principle: whenever a node is created or incurs over/under-flows, its size is re-computed using the current statistical information. In the sequel, we demonstrate this by discussing adaptive R-trees optimized for window queries.

To avoid excessively complex equations, we focus on the so-called *quadratic window query*, whose extents along the x- and y- dimensions have equal lengths q_L. The application to queries with arbitrary extents in higher dimensions is straightforward. Two parameters are needed to describe a uniform data distribution: (i) the number N of spatial objects, and (ii) their *density D*. Specifically, the density of a set of rectangles is defined as the average number of rectangles that contain a given point in the data space. Equivalently, D can be expressed as the ratio of the sum of the areas of all rectangles over the area of the data space. The performance of R-trees has been very well studied (see [PSW95, TSS00]). The following model [TSS00] gives the number of leaf node accesses for uniform data distribution (where f_0 is the fanout, D_0 the density, and N_0 the number of leaf nodes):

$$NA_{LEAF}(q_L) = N_0\left(\sqrt{\frac{D_0}{N_0}} + q_L\right)^2, \text{ where}$$

$$D_0 = \left(1 + \frac{\sqrt{D} - 1}{\sqrt{f_0}}\right)^2, \text{ and } N_0 = \frac{N}{f_0} \quad (4\text{-}1)$$

Hence the total cost (response time) of processing the leaf level is (p_0 is the size of leaf nodes):

$$TIME_{LEAF}(q_L) = N_0\left(\sqrt{\frac{D_0}{N_0}} + q_L\right)^2 (T_{SK} + p_0 \cdot T_{TRF} + f_0 \cdot T_{EVL}) \quad (4\text{-}2)$$

Obtaining the optimal node size, however, is not straightforward because the solution of the above derivative requires numerical approaches that are too expensive to compute in real-time. Instead of computing the derivative, we adopt the algorithm in Figure 4.1 to find the optimal size directly with equation (4-2). The algorithm starts with an initial size $p=p_{GUESS}$, and then refines it iteratively by modifying p towards minimizing the access time (line 7). This procedure is repeated until the optimal size has been found or a certain time limit expires (e.g., 0.1 seconds), after which the current value of p is returned. Since the optimal node sizes are usually integers below 100, this algorithm finds the optimal solution very quickly, as proved in our experiments.

Unlike range queries in B-trees, answering a window query usually requires visiting multiple nodes at all levels (except the root) of the R-tree. Hence sizes of non-leaf nodes should also be optimized to improve performance. Similar to the leaf level, the number NA_i of node accesses at level i ($1 \le i \le h-1$) can be written as (f_i is the fanout, D_i the density, and N_i the number of level i nodes):

$$NA_i(q_L) = N_i\left(\sqrt{\frac{D_i}{N_i}} + q_L\right)^2, \text{ where}$$

$$D_i = \left(1 + \frac{\sqrt{D_{i-1}} - 1}{\sqrt{f_i}}\right)^2, \text{ and } N_i = N_{i-1}/f_i$$

The processing cost at level *i* is similar to equation (4-2), and the optimal node size can be found also with the algorithm in Figure 4.1.

```
Algorithm Find_Optimal_Node_Size (TIME(p): the time function with node size p as parameter)
1.  p = p_GUESS /*an initial guess value*/
2.  t_0 = TIME (p) /*the access time if the node size is p*/
3.  set δp to some positive value
4.  do {
5.      p = p+δp
6.      t_1 = TIME (p)
7.      if t_1 > t_0 then δp = –δp/2   /*if t_1 ≤ t_0 then do nothing*/
8.      t_0 = t_1
9.  }while (δp≠0 && time limit has not expired)
10. return p
end Find_Optimal_Node_Size
```

Figure 4.1: Algorithm for finding the optimal node size

Extending the above analysis to general query distribution deservers further elaboration, because, unlike adaptive B-trees, the problem is no longer equivalent to optimizing the query with the expected length $Exp(q_L)$. To illustrate this, assume the existence of arrays $q_L[]$ and $pr[]$ (with similar semantics to those in section 3.1). The expected processing cost at the leaf level is:

$$\begin{aligned} &TIME_{LEAF}(q_L) \\ &= \sum_{i=1}^{t} \left[pr[i] \cdot N_0 \left(\sqrt{\frac{D_0}{N_0}} + q_L[i] \right)^2 \cdot (T_{SK} + p_0 \cdot T_{TRF} + f_0 \cdot T_{EVL}) \right] \quad (4\text{-}3) \\ &= \left(\frac{D_0}{N_0} + 2\sqrt{\frac{D_0}{N_0}} \sum_{i=1}^{t} pr[i] \cdot q_L[i] + \sum_{i=1}^{t} pr[i] \cdot (q_L[i])^2 \right) \cdot N_0 (T_{SK} + p_0 \cdot T_{TRF} + f_0 \cdot T_{EVL}) \\ &= \left(\frac{D_0}{N_0} + 2\sqrt{\frac{D_0}{N_0}} Exp(q_L) + Exp(q_L^2) \right) \cdot N_0 (T_{SK} + p_0 \cdot T_{TRF} + f_0 \cdot T_{EVL}) \end{aligned}$$

Therefore, instead of keeping $q_L[]$ and $pr[]$, it suffices to maintain the total number of queries, the sum of query lengths, and the sum ($sum\_q_L^2$) of square lengths (i.e., the area of the query). Similar discussion applies to higher levels. To extend the analysis to arbitrary distributions, we maintain a histogram where each bin corresponds to the cell of a regular grid. The following information is stored in each bin$_j$ ($1 \le j \le num_{bin}$): (i) number (n_j) of objects in bin$_j$, (ii) total number of queries ($total\_q_j$) whose extents intersect that of bin$_j$ (iii) sum (sum_q_{Lj}) of length, and (iv) sum ($sum\_q_L{}^2{}_j$) of areas of queries in bin$_j$. For a query (as in Figure 4.2) that crosses the boundary of bin$_j$, $\sqrt{q_1 \cdot q_2}$, and $q_1 \cdot q_2$ are added to sum_q_{Lj} and $sum\_q_L{}^2{}_j$ respectively.

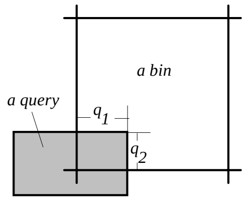


Figure 4.2: A query that crosses the boundary of a bin

The cost of processing in bin$_j$ is represented as in (4-3) with the following modifications (recall that D_0 is computed from D as in equation 4-1):

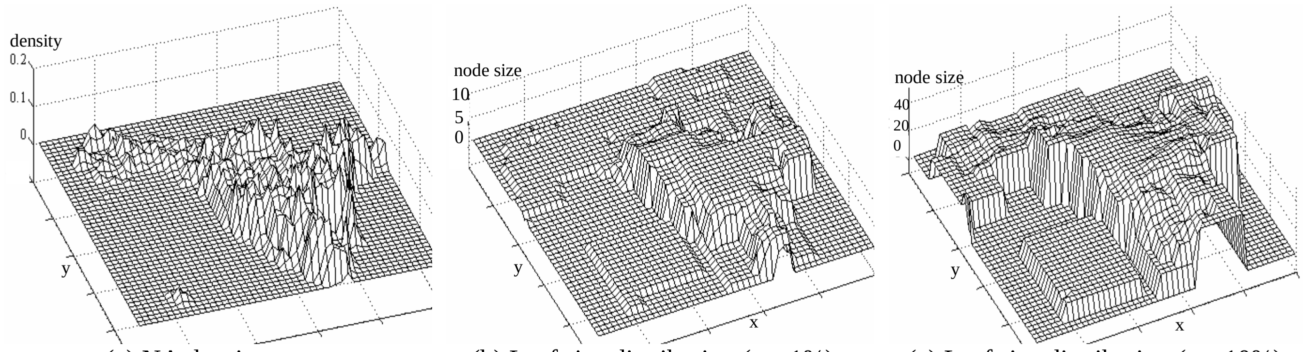
$$D = sum\_q_L{}^2{}_j / n_j, \; Exp(q_L) = sum\_q_{Lj} / n_j$$

$$N_0 = n_j \cdot num_{bin} / f_0, \; Exp(q_L{}^2) = sum\_q_L{}^2{}_j / n_j$$

The optimal node size (for all levels) can also be found with the algorithm in Figure 4.1.

To demonstrate the structures of adaptive R-trees, we use the density map [TSS00] of a real dataset that contains MBRs of 560K roads in North America (Figure 4.3a). Figure 4.3b shows the size distribution of leaf nodes of an adaptive tree optimized for quadratic queries with q_L=1% (i.e., the query area covers 0.01% of the universe). Notice that nodes that correspond to low density have the minimum size 1 while those covering denser areas are larger. Figure 4.3c shows the node distribution for the tree optimized for q_L=10%, where the node sizes increase considerably.

Our implementation is based on R*-trees [BBKS90]



(a) NA density map (b) Leaf size distribution (q_L=1%) (c) Leaf size distribution (q_L=10%)

Figure 4.3: Visualization of the structure of adaptive R-trees

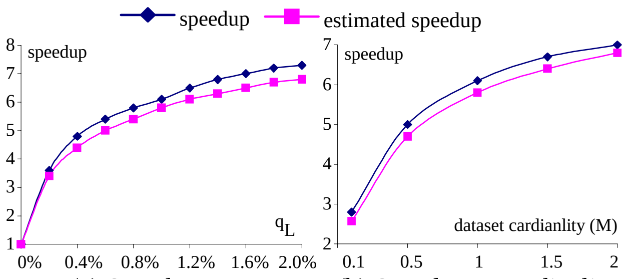
because they are considered the most efficient R-tree variation. The update algorithms of adaptive R*-trees are similar to those of B-trees. It is worth mentioning only the following basic differences: (i) the minimum node utilization is set to 40% [BBKS90] (which is necessary for the efficiency of the split algorithm), (ii) merging is never performed because in R*-trees entries in a node that underflows are re-inserted and (iii) there do not exist pointers between sibling nodes. Finally, the bulkloading and partial rebuilding techniques also extend to R-trees with straightforward modifications.

5. Experiments

In this section we demonstrate the efficiency of adaptive index structures with extensive experimental evaluation. The adopted values for T_{SK}, T_{TRF} are 10ms and 1ms/Kbytes respectively, which are close to the specifications of recent products from renowned hard disk vendors [Web]. To estimate T_{EVL}, we assume each entry evaluation consumes 1000 clock ticks, leading to around 1μs for a 1GHz CPU. The page size (also the size of a node in conventional B- and R-trees) is set to 1K bytes in all cases. With this size, the maximum number of entries in a page is 125 (50) for both B- (R-) and adaptive B- (R-) trees. We start with results on adaptive B-trees and then discuss R-trees.

The first set of experiments explores the effects of data and query parameters on performance. We use synthetic datasets (cardinality N 100K-2M) that contain records whose search keys are uniformly distributed in the universe (i.e., a unit line segment). The sizes of the resulting B-trees and adaptive B-trees are similar (around 11.5 Mbytes for 1M objects), with adaptive trees being slightly smaller. Then we apply query workloads, each consisting of 500 queries (also uniformly distributed in the universe) with the same query length q_L. The query length (selectivity) of different workloads varies from 0 (workloads consisting solely of equality selection queries) to 2% of the universe. Notice that, since both data and query distributions are uniform, all nodes in the adaptive trees have the same size.

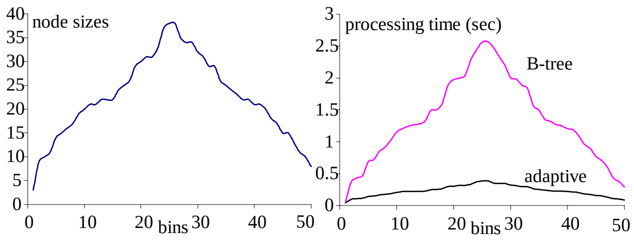
Figures 5.1a shows the speedup (i.e., the total cost of a workload on the B-tree over that on the adaptive tree) as a function of q_L for the dataset with N=1M records. Figure 5.1b illustrates the speedup as a function of N for q_L=1%. Adaptive B-trees are clearly better in all cases except for q_L=0% (where they degenerate to conventional B-trees). As expected, the speedup increases with q_L and N, since more records are retrieved, which, in turn, increases the number of random accesses for conventional B-trees. In addition, the diagrams contain the estimated speedup obtained from equation (3.3-2), which is very close to the actual value. Further, as discussed in section 3.4, the speedup tends to converge to a value, estimated by (3.3-3), for large query extent or high cardinality.



(a) Speedup vs q_L (b) Speedup vs cardinality

Figure 5.1: Effects of data and query parameters

The next experiment simulates a situation where the dataset (1M records) is uniform, but queries have different lengths in different parts of the data space. Statistical information is kept in a histogram consisting of 50 bins. Each bin is associated with an expected query length, which follows a gaussian distribution in the range [0, 2%]. The bins at both ends of the universe correspond to the shortest (i.e., most selective) queries, while bins at the centre to the longest ones. Figure 5.2a shows the sizes of leaf nodes of the respective adaptive tree in different bins (the x-axis corresponds to the 50 bins). The sizes follow the expected query distribution: nodes in bins at both ends are the smallest while the one at the centre is the largest (up to 37 pages). Observe that the optimal node size varies significantly with the query length, which indicates that using a single node size throughout the tree cannot optimize query performance.

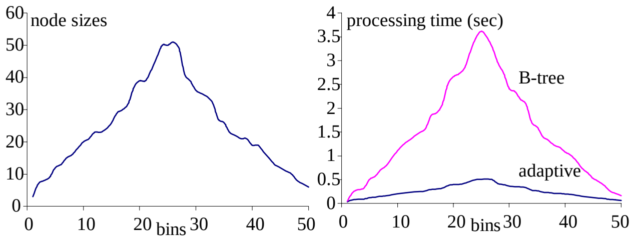


(a) Node size vs. bin (b) Query cost vs. bin

Figure 5.2: Gaussian workload and uniform dataset

Then we create a workload of 5000 queries (to ensure that each bin receives enough queries) according to the expected distribution. Figure 5.2b compares the average performance of B- and adaptive trees within each bin. It is clear that the conventional tree is comparable to the adaptive version only in bins where queries have very short lengths (close to 0). Observe (for all experiments) that although a large node size (e.g., 8K bytes) for conventional B-trees will decrease the speedup of long queries, the gain with respect to short ones (e.g., equality selections) will increase accordingly; thus, the overall benefit of the adaptive structure will remain more or less the same.

Figure 5.3 shows results of a similar experiment for a dataset (cardinality 1M) with records whose search keys follow a gaussian distribution. The only difference from Figure 5.2 is that the node size (and the speedup) in the central bins is now larger due to the higher density in the corresponding part of the data space. In the sequel, we only show results for gaussian workloads on uniform datasets.



(a) Node size vs. bin (b) Query cost vs. bin

Figure 5.3: Gaussian workload and Gaussian dataset

To evaluate the effect of buffering, we introduce caches with sizes up to 50% of the trees (1M records). As shown in Figure 5.4, the speedup declines with the buffer size, which is expected because caches decrease the disk access probability. Notice, however, that significant speedup is achieved even for buffers containing 50% of the tree.

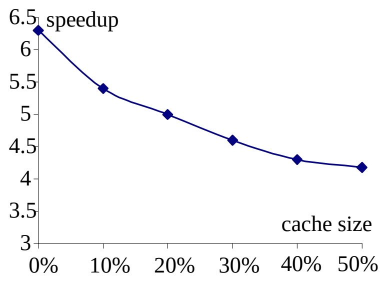


Figure 5.4: Speedup vs cache size

To study the effect of updates, we perform a "mixed" workload of 10,000 queries and updates (involving equal numbers of insertions and deletions). Figure 5.5 shows the speedup as a function of the percentage of updates in the workload. The performance gain is considerable in all but the cases where most operations are updates. The update cost of the adaptive tree is slightly higher since it needs to (sequentially) read multiple pages for a node and, potentially, relocate some pages.

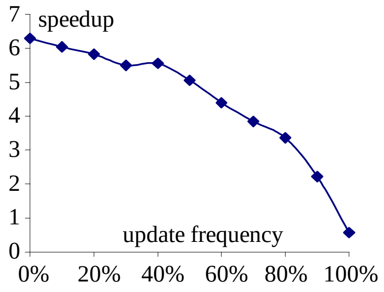
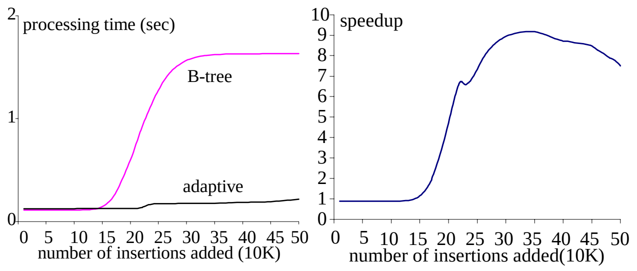


Figure 5.5: Speedup vs update frequency

In order to demonstrate the performance deterioration after bulkloading, we create a dataset with 500K uniform records, and bulkload a B- and an adaptive B-tree. Then, we perform another 500K insertions. Figure 5.6a shows the query cost of the two structures as a function of the number of insertions (5 means 50K insertions are performed and so on). Before 150K insertions both trees have similar performance because most accesses are sequential. After that, the B-tree starts to incur node splits that break the sibling adjacency (as discussed in section 3.5), and its performance deteriorates very quickly. The cost increase of the adaptive tree, on the other hand, is very slow because it adjusts node sizes to reduce page accesses. Figure 5.6b shows the speedup as a function of the number of insertions.



(a) Performance (b) Speedup

Figure 5.6: Deterioration after bulkloading

Finally we test the generality of the proposed techniques by comparing adaptive R*- trees with their conventional counterparts using the NA dataset (described in section 4). We apply workloads of 500 (quadratic) window queries whose distribution in the space follows that of the data (in order to avoid meaningless queries in empty areas). The histogram contains 10×10 bins. Figure 5.7 shows the cost of R*- and adaptive R*-trees for workloads with q_L ranging from 0% (i.e., point queries) to 10% (i.e., covering 1% of the space). Similar to B-trees, the conventional tree is comparable to the adaptive one only for small queries, and the speedup increases with the query size.

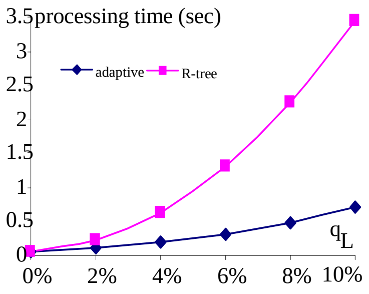


Figure 5.7: Total cost vs q_L for R-trees

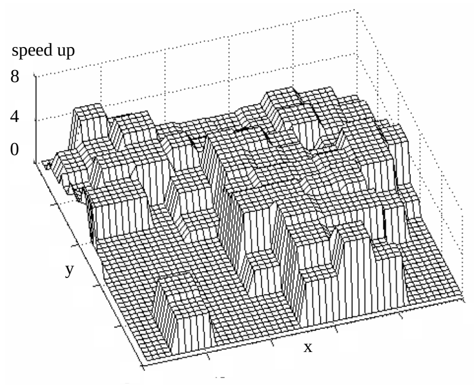


Figure 5.8: Speedup in different areas

Figure 5.8 shows the speedup for queries (q_L=5%) at different positions in the data space. Observe that larger speedup is achieved in high-density areas. Figure 5.9, measures the processing time (q_L=5%) as a function of the cache size (depicted as the percentage of the tree). Large buffers favour both trees and the adaptive tree achieves significant speedup in all cases. Finally, results about update frequency and performance deterioration of bulkloading are omitted because they are similar to those of B-trees.

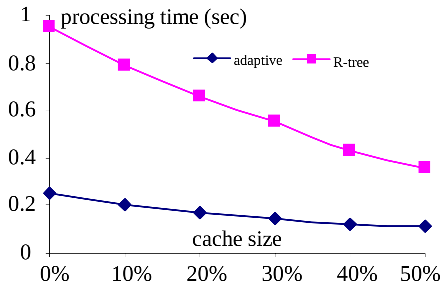


Figure 5.9: Total cost vs cache size for R-trees

6. Conclusion

While the node size has a significant effect on the performance of database indexes, in practice the decision is usually made in an ad-hoc manner. This is because a good choice for the size depends on several data and query parameters, which are often unavailable in advance and highly dynamic. In this paper we introduce the concept of adaptive index structures, which dynamically adapt their node sizes (according to these parameters) to minimize the query cost. We also propose a general framework for converting traditional structures to adaptive versions, through a set of update and bulkloading algorithms. The only requirement for our methods is the existence of analytical models that estimate the number of node accesses. Such models have been proposed for most popular structures rendering our framework directly applicable to them.

Notice that even if an optimal node size for a conventional index can be determined in advance, this size would apply to the whole structure. On the other hand, the proposed indexes permit variable node sizes that follow the query characteristics in different parts of the data space. Given the ubiquitous use of histograms in modern databases, adaptive structures can take advantage of statistical information to accelerate performance. Analytical and experimental evaluation confirms that adaptive indexes outperform conventional counterparts significantly in a wide range of scenarios.

Furthermore, this work also initiates numerous research problems. For example, in this paper the node size is optimized without buffers. Performance in the presence of buffers can be further improved if the node size is optimized considering the cache size. Another interesting direction is to investigate the application of the techniques to structures, for which there do not exist any cost models. Sampling approaches, for example, may be used in this case to find a good (although perhaps not optimal) node size. Finally, the methods can be extended to other methodologies (e.g., hashing) that involve disk access issues.

Acknowledgements

This work was supported by grants HKUST 6081/01E and HKUST 6070/00E from Hong Kong RGC.

References

[APR99] Acharya, S., Poosala, V., Ramaswamy, S. Selectivity Estimation in Spatial Databases. *ACM SIGMOD*, 1999.

[BCG01] Bruno, N., Chaudhuri, S., Gravano, L. Stholes: A Multidimensional Workload-Aware Histogram. *ACM SIGMOD*, 2001.

[BKK96] Berchtold, S., Keim, D. A., Kriegel, H.-P. The X-tree: An Index Structure for High-Dimensional Data, *VLDB*, 1996.

[BKSS90] Beckmann, N., Kriegel, H., Schneider, R., Seeger, B. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. *ACM SIGMOD*, 1990.

[CR94] Chen, C., Roussopoulos, N. Adaptive Selectivity Estimation Using Query Feedback. *ACM SIGMOD*, 1994.

[G84] Guttman, A. R-trees: A Dynamic Index Structure for Spatial Searching, ACM SIGMOD, 1984.

[GLR00] Ganti, V., Lee, M., Ramakrishnan, R. Icicles: Self-Turning Samples for Approximate Query Answering. *VLDB*, 2000.

[GM98] Gibbons, P., Matias, Y. New Sampling-Based Summary Statistics for Improving Approximate Query Answers. *ACM SIGMOD*, 1998.

[HS92] Haas, P., Swami, A. Sequential Sampling Procedures for Query Size Estimation. *ACM SIGMOD*, 1992.

[IP95] Ioannidis, Y., Poosala, V. Balancing Histogram Optimality and Practicality for Query Result Size Estimation. *ACM SIGMOD*, 1995.

[PSW95] Pagel, B.U., Six, H.W., Winter, M. Window Query-Optimal Clustering of Spatial Objects. *ACM PODS*, 1995.

[SRF87] Sellis, T., Roussopoulos, N. Faloutsos, C.: The R+-tree: a Dynamic Index for Multi-Dimensional Objects, *VLDB*, 1987.

[TSS00] Theodoridis, Y., Stefanakis, E., Sellis, T. Efficient Cost Models for Spatial Queries Using R-trees. *IEEE TKDE*, 12(1), pp. 19-32, 2000.

[TPZ01] Tao, Y., Papadias, D., Zhang, J. Cost Models for Overlapping and Multi-Version Structures. Technical Report HKUST01. Available at http://www.cs.ust.hk/~dimitris/

[WAA02] Wu, Y., Agrawal, D., El Abbadi, A. Query Estimation by Adaptive Sampling. *IEEE ICDE*, 2002.

[WAA01] Wu, Y., Agrawal, D., El Abbadi, A. Applying the Golden Rule of Sampling for Query Estimation. *ACM SIGMOD*, 2001.

[Web] http://www.storage.ibm.com/hdd/index.htm