# Enhancement of Data Aggregation Using A Novel Point Access Method

HUNG-YI LIN[1], RONG-CHANG CHEN[1], SHIH-YING CHEN[2]
[1]Department of Logistics Engineering & Management
[2]Department of Computer Science and Information Engineering
National Taichung Institute of Technology
129, Sanmin Rd., Sec. 3, Taichung
TAIWAN, R.O.C.
{linhy, rcchens, sychen}@ntit.edu.tw

*Abstract:* -The $B^+$-tree and its variants have been reported as the good index structures for retrieving data. Database systems frequently establish the $B^+$-tree style indices for fast access to data records. However, traditional $B^+$-tree index could be a performance bottleneck because of its inflatable hierarchy. Many works focus on improving indexing techniques. In fact, the optimization of data organization inside index nodes is the most critical factor to improve retrieval quality. Some handles like pre-partition of data space, node splitting by force, node splitting with unbalanced partition, and node splitting upon overflow loading always burden index structures with plenty of storage space and building overhead. In this paper, we propose a new index scheme to highly aggregate the external structure in a $B^+$-tree. It also adopts a better splitting policy to completely remove the suffering from data insertion orders. Our new index technique can compress data records in leaves and in turn reduce index size to improve query performance. In addition, the entire index's space utilization is promoted to a higher level; thereby the index's space requirement becomes smaller and easily resides in memory.

*Key-Words:* - $B^+$-tree, Index structure, $B$*-tree, Databases, Data aggregation, Node splitting.

## 1 Introduction

As the price of memory continuously becomes cheaper, many of database tables and indices are easily placed in main memory. Traditional bottleneck of disk access is replaced by memory access [5]. It has been revealed that commercial DBMSs spend half the execution time on memory access when a whole database fits in memory [2]. As a result, today's database applications are becoming increasingly compute and memory bound. Many works address the techniques focusing on minimizing memory latencies when the database is accessed. The latency time in retrieving data from the memory resided with a huge index is considerably time-consuming. In fact, compactly coordinating the data arrangement in a database index is essential toward the acquisition of memory performance.

On the other hand, modern processors make use of processor cache to alleviate memory bound. Modern processors have up to several megabytes of SRAM as the cache, which can be accessed in one or two process cycles while one memory access costs tens of processor cycles. Typically, the index's node size is equal to the cache line size in order to minimize the number of cache misses. However, the low space utilization (50%~70%) in traditional index nodes can not provide sufficient data for every cache allocation, which makes the processor need more cache accesses to embrace the requested data. For promoting the data quality for every cache allocation, aggregating data at leaf nodes and increasing the space utilization of external structures are necessary when an index scheme is developed.

Ailamaki et al. [1] have reported that faster processors do not improve database system performance to the same extent as scientific workloads. The result is that from the perspective of the processor, memory has been getting slower at a dramatic rate. This makes it increasingly difficult to achieve high processor efficiencies. For narrowing the performance gap between processors and memory, many works propose the techniques focusing on minimizing memory latencies, such as concurrency control algorithms [13, 19], cache-conscious algorithms [17], and prefetching $B^+$-trees [7]. These techniques have modified core database algorithms to make them more hardware friendly. In this paper, our focus returns to the design logics of index schemes. Memory access performance and

This research was supported by Nation Science Council of ROC under Grant No. 96-2622-E-025-003-CC3.

cache misses is improved by our compressed index structure: compressed $B^+$-tree. The techniques mentioned previously in improving memory latencies can collaborate with our scheme to gain the better performance.

## 2 Overviews of Previous Works

Many traditional index schemes focus on improving index techniques. One of the common policies is to adopt pre-partitions of data space [6, 11, 15, 19] before processing the index scheme. Applying space pre-partitions is difficult to meet the requirements of real data distribution since the data sequence used to develop an index can not be determined before processing an index scheme. Pre-partitioning policies are incapable of maintaining high data aggregation in the resulting external structures, especially in case of high data skewness. Other policies adopt immediate node splitting when an emerging data intends to join a full target. Node splitting policies can adopt unbalanced partition [10, 18], or alternatively, balanced partition [8, 14, 16, 21]. Splitting policies are the typical examples severely suffering from the problem of data insertion order. Based on the design issue of node splitting policies, the neighboring data already arranged together are very likely separated by consequent data. Such scenario frequently fragments the data space. Since the handle of splitting is irreversible, the departure of neighboring data can not recover except executing many deletions and reinsertions. These traditional indices resided in memory can not supply caches with high data retrieval quality. *R**-tree [3] and extended *CIR*-tree [12] use the re-insertion techniques to reorganize parts of improper data arrangement in leaves. Although less splits are performed, the insertion routine is called more often in restructuring an R*-tree or an extended *CIR*-tree. The problem of unnecessary splits suffered from data insertion order is not completely eliminated by their methods. The cache quality and the cache misses still stay at a low level.

Instead of improving index techniques, many efforts focus on improving the index access methods. Hankins et al. [9] developed an analytical model based on the fundamental components of the search process. This model is validated with experimental results and demonstrating that using node sizes much larger than the cache lines can result in better search performance for the $CSB^+$-tree [17]. Prefetching $B^+$-trees [7] use prefetching to effectively create wider nodes than the natural data transfer size. These wider nodes reduce the height of the $B^+$-tree and accelerate operations of searches and range scans on $B^+$-tree indices. In fact, the essential idea of $B^+$-tree's indexing technique is invariant in [7, 9]. Their performance improvements are not gained from taking full advantage of the architectural innovations.

A new concurrent $B^+$-tree algorithm [13] provides high concurrency to conquer the performance bottleneck. A bottom-up tree restructuring method using key-range indicators contained in each leaf node always preserves a semantically consistent state of the $B^+$-tree. Although a concurrent $B^+$-tree can execute efficient range searches and is suitable for the high performance transaction processing. The time and space efficiencies of index maintenance are respectively deteriorated by the tree restructuring technique and extra key-range indicators.

Because $B^+$-tree and its variants adopt the similar index policies, we only concentrate on the performance difference between conventional $B^+$-tree and our compressed $B^+$-tree in this paper. We analyze the effect of our proposed index scheme and make the following contributions:

- Our insertion algorithm breaks the dilemma derived from data insertion order.
- The storage utilization is significantly increased and the storage requirement is remarkably reduced.
- No restructuring handle is required; thereby no additional overhead is requested.
- The more condensed index resides in the memory. Caches are supplied data with higher quality and quantity. Cache misses and memory bound are improved.

The rest of this paper is organized as follows. Section 3 demonstrates our design logics by an example. Section 4 describes and explains all related algorithms in building a compressed $B^+$-tree. Section 5 analyses the performance of a compressed $B^+$-tree comparing with traditional $B^+$-tree. Section 6 employs the real geographical data for competitive studies. Several measurements used to evaluate various performances are given in this section. In Section 7, we summarize our key results.

## 3 New Index Scheme

The motivation of our new index technique comes from the exploitation of the unused space around full targets. The vacant space available in a full target's predecessor or successor can be taken for dispersing the upcoming overload in the target. *B**-tree proposed by [4] had pointed out increasing storage utilization has the impact of speeding up the search since the height of the resulting tree is smaller. The index structure of *B**-trees is based on the structure of *B*-trees which employs a local redistribution scheme to delay splitting until a full target's next sibling is also fully loaded. This scheme guarantees that 2 full nodes are divided into 3, so the utilization of each node is at least 2/3 full (i.e. 66%). In this study, although our idea has some design similar to *B**-trees, we address three major characteristics that differs significantly from *B**-trees.

1. *B**-tree is a variant of *B*-tree that attempts to improve the storage utilization of *B*-tree while our intention is on improving $B^+$-tree.
2. A supervisal mechanism is developed to evaluate the degree of data aggregation in the external index structure. This mechanism also plays the criterion to pick the suitable sibling (target's predecessor or successor) and decide the optimal data quantity for executing local redistribution.
3. Since our local redistribution scheme involves at most 3 nodes, it guarantees that 3 full nodes are divided into 4. The utilization of each node is at least 75% which outperforms *B**-tree's 66%.

We proceed to illustrate our design by the following example. Suppose a data sequence of {5, 8, 1, 4, 3, 12, 9, 6, 15, 7} is assigned to be organized by an index. To exemplify, the maximum number of node capacity is assumed to be 5. Fig. 1(a) and 1(b) shows the building procedures induced by our proposed scheme and the conventional index scheme, respectively. We state the relative operations between the conventional scheme and our scheme step by step as follows.

Step 1. Initially, data 5, 8, 1, 4, and 3 are inserted into the root in turn. Then, datum 12 is going to join the full root.

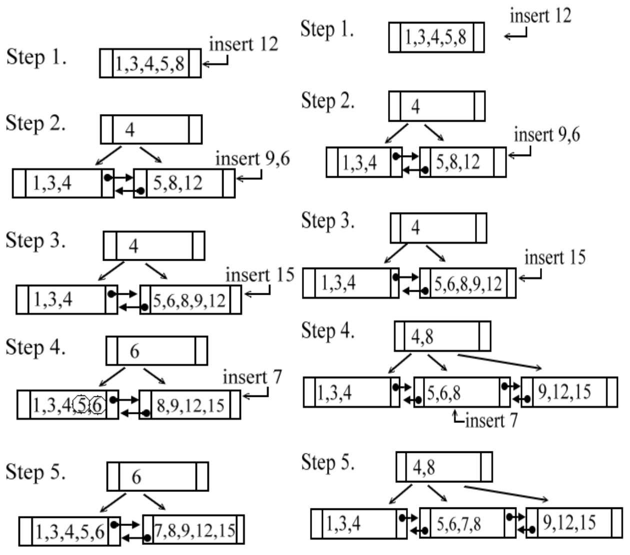
Step 2. The root's splitting grows the index with one more level. Next, data 9 and 6 join the right leaf in turn and make the target full.

Step 3. Again, datum 15 is joining the right leaf. In Fig. 1(a), the full target appeals to its left sibling. Whereas, in Fig. 1(b), one leaf splitting is activated.

Step 4. In Fig. 1(a), data 5 and 6 in the right leaf are removed to the left leaf and the relative upper entry is modified accordingly. The spared space make datum 15 accommodate in the target. In Fig. 1(b), the third leave is generated for datum 15. After that, datum 7 is the last insertion.

Step 5. Finally, our scheme allocates 3 nodes for ten data while the conventional index scheme allocates 4 nodes.

We note that, in Step 4, the migration of data 5 and 6 to the left sibling causes the data variance at leaf level smaller than that only moving datum 5. Namely,
$$s^2(1,3,4,5,6) + s^2(8,9,12,15) < s^2(1,3,4,5) + s^2(6,8,9,12,15).$$
To evaluate the data aggregation within a certain leaf, the compactness of data organization is critical as well. So, not only data variance but also data density must be taken into account. A sufficient data quantity with a small data variance is the necessary condition for high data aggregation. We term $\Delta(N_i)$ the *data aggregation* of node $N_i$ which is formulated as $\frac{U(N_i)}{s^2(N_i)}$, where $U(N_i)$ is the space utilization of node $N_i$.



(a) (b)

Fig. 1 The constructing procedures of (a) a compressed index and (b) a conventional index.

Suppose two neighbor nodes *A* and *B* (*B* is next to *A*) are involved for data redistribution. We assume there are originally *n* and *m* data entries contained in *A* and *B*. Accordingly, the data aggregation of them is denoted as $\Delta_n(A) + \Delta_m(B)$. One or several seems to be likely move from *A* to *B*, or *B* to *A*. In case of one data is removed from *A* to *B*, the data aggregation becomes $\Delta_{n-1}(A_R^-) + \Delta_{m+1}(B_L^+)$ , where $A_R^-$ represents node *A* takes away its rightmost datum and $B_L^+$ represents node *B* accommodates this datum at its left side. If necessary, the second datum may be taken away as well from $A_R^-$ and pushed into $B_L^+$ and so forth. As a result, the data aggregation becomes $\Delta_{n-k}(A_R^-) + \Delta_{m+k}(B_L^+)$ , where $1 \le k \le M - m$ and $M$

is the node capacity. The optimal number $k$ for data redistribution depends on selecting the minimum from all possible combinations. On the other hand, in case of migrating the leftmost data from $B$ to $A$'s right side, the value of $k$ is decided by minimizing $\Delta_{n+k}(A_R^+) + \Delta_{m-k}(B_L^-)$ . The detailed calculation process is given in the Appendix.

For a comprehensive study of the problem derived from insertion orders, all possibilities ($10 \times 9 \times \cdots \times 2 \times 1$=3,628,800) of insertion orders for the 10 data {5, 8, 1, 4, 3, 12, 9, 6, 15, 7} are fully generated for analysis. The effect of data insertion orders on the external structure is listed in Table 1. Using the same node capacity, no matter what data sequence is adopted; our proposed scheme organizes these data into the same index as shown in the result of Fig. 1(a). However, a traditional scheme results in four cases with different data classifications according to different data sequences. The term $\Omega$ in Table 1 means the entire data aggregation of an index which is computed by collecting all individual data aggregation $\Delta(N_i)$ of leaves. That is,

$$\Omega = \sum_{i}^{\text{all leaves}} \Delta(N_i) .$$

Our proposed scheme only results in the data classification as given in the first case of Table 1 which possesses the 100% external space utilization and the corresponding $\Omega$ value is the largest among all cases. Unfortunately, under a low probability of 1.38%, a traditional scheme can achieve the data classification as Case 1. With a high probability of 98.62% in this example, a traditional scheme indexes these 10 data in inflatable hierarchies (case 2~4) whose data aggregations are smaller than that of case 1. In addition, the external space utilization in Cases 2~4 is only 67%. This example validates the fact that traditional index schemes frequently suffer from improper data sequences.

## 4 Structure and Algorithms

The construction of a tree hierarchy is based on organizing the leaf nodes. Thus, the policy in coordinating data inside leaf nodes and the strategy in allocating new leaf nodes are so critical for the consequent entry arrangement in the internal nodes. The internal nodes of an index serve as a directory for fast data retrieval. The efficacy of such directory significantly impacts the maintenance and query performance of the index structure.

The base structure of the compressed $B^+$-tree is similar to that of the $B^+$-tree. The leaf nodes preserve the data which are classified by the entries preserved in the internal nodes. In order to support our new indexing scheme, each compressed $B^+$-tree's leaf contains a pointer to its right sibling and a pointer to its left sibling (the pointer is non-null if the sibling exists). The major difference between compressed $B^+$-tree and $B^+$-tree is the leaf splitting policy. A compressed $B^+$-tree appeals to the vacant space found in the right sibling and even the left sibling of the arrived target leaf encountering a new insertion. This leaf collaboration delays leaf splitting as far as possible. We assert that the overhead spent in a data migration is far lower than that spent in activating a split. Splitting operations should be invoked in the circumstance that no alternative handle can support the proper processing.

Suppose $M$ is the maximum number of node capacity. The internal node of a compressed $B^+$-tree is represented as [$p_0$, $k_1$, $p_1$, … $k_i$, $p_i$], where $p_i$ is the pointer referring to the node located at the next level and $k_i$ is the key used as the boundary value between two groups of data, $1 \le i \le M$ . The external node of a compressed $B^+$-tree is designed as [$\downarrow_L$, $d_1$, …, $d_i$, $\downarrow_R$]. Two pointers $\downarrow_L$ and $\downarrow_R$ link to the predecessor and successor leaves, respectively. The key $d_i$ preserves the indexed data, $1 \le i \le M$ . The following notations are used through our algorithms.

- $T$: The root of the compressed $B^+$-tree.
- $N$, $NN$, $O$: A node of the compressed $B^+$-tree.
- $parent(N)$: The parent of node $N$.
- $load(N)$: The load of node $N$.
- $k$, $k_1$, $k_2$, $k_3$: The key in a node.
- $S$, $S_1$, $S_2$: The sets used to collect keys.

Table 1 Four cases of data classifications organized by the traditional index scheme.

| Case | #leaves | Data contents in leaves | $\Omega$ ($\times 10^{-2}$) | Frequency | Probability |
|---|---|---|---|---|---|
| 1 | 2 | (1,3,4,5,6)、(7,8,9,12,15) | 8.7 | 50,112 | 1.38% |
| 2 | 3 | (1,3,4)、(5,6,7,8)、(9,12,15) | 7.1 | 30,798 | 0.85% |
| 3 | | (1,3,4)、(5,6,7)、(8,9,12,15) | 7.6 | 63,594 | 1.75% |
| 4 | | (1,3,4,5)、(6,7,8)、(9,12,15) | 7.2 | 3,484,296 | 96.02% |
| Total | | | | 3,628,800 | 100% |

**Algorithm Insert**(*T*, *I*)
**Input**: The root *T* of a compressed $B^+$-tree and the data *I* to be inserted.
**Output**: The root of the new compressed $B^+$-tree.

```
1. $N \leftarrow T$.
2. While $N$ is a non-leaf node,
     If $I \le N.k_1$,
       $N \leftarrow N.p_0$
     else, for a specific key $N.k_s$ such
       that $N.k_s < I \le N.k_{s+1}$, $s \le load(N)$
       $N \leftarrow N.p_s$.
3. If $N$ is under-full, then join $I$ in $N$ and increase
     $load(N)$ by one.
4. If $N$ is full, let $N'$ contain the entries of $N$ plus $I$
     and select one of the following cases:
   Case a: $load(N.\downarrow_L) < M$ and $load(N.\downarrow_R) < M$
       For three leaves $N.\downarrow_L$, $N'$, and $N.\downarrow_R$, if
       removing the largest entry from $N'$ to $N.\downarrow_R$
       can result in the smaller total data
       aggregation ($\Omega$) than that removing the
       smallest entry from $N'$ 'to $N.\downarrow_L$, then data
       migration is invoked between $N'$ and $N.\downarrow_R$.
       Inversely, data migration is invoked between
       $N.\downarrow_L$ and $N'$.
   Case b: $load(N.\downarrow_L) = M$ and $load(N.\downarrow_R) < M$
       Data migration is invoked between $N'$ and
       $N.\downarrow_R$
   Case c: $load(N.\downarrow_L) < M$ and $load(N.\downarrow_R) = M$
       Data migration is invoked between $N.\downarrow_L$
       and $N'$.
   Case d: Two siblings of $N$ are both full.
       Invoke Split($N$, $I$)
5. Return $T$.
```

In the Case a of insertion algorithm, $\Delta(N.\downarrow_L) + \Delta((N')^-_R) + \Delta((N.\downarrow_R)^+_L)$ and $N.\downarrow_L$ + $\Delta((N')^-_L) + \Delta(N.\downarrow_R)$ are evaluated and compared to determine the direction of data migration. In addition, for retaining their data aggregation at a high level, not only one but several data may be adjusted between leaves in the handles of Cases a, b, and c.

As mentioned in Section 3, a full target in a *B**-tree only appeals to the next sibling when encountering a new insertion. It is easy to simplify our listed cases in the insertion algorithm to meet the basic ideal of *B**-tree. In this study, we term the index structure proposed by our design as *compressed* $B^+$*-tree* (denoted by $CB^+$*-tree*). The simplified design only appealing to the next sibling is termed as *compressed* $B_R^+$*-tree* (denoted by $CB_R^+$*-tree*).

**Algorithm Split**(*O*, *I*)

```
1. $S \leftarrow$ Sort all keys in $O$ plus $I$ increasingly.
2. If $O$ is a leaf,
   a. $k_1 \leftarrow$ the $\lceil (M+1)/2 \rceil$-th largest key in $S$.
   b. Allocate a new node $NN$.
   c. $O \leftarrow$ the first $\lceil (M+1)/2 \rceil$-th keys in $S$. $NN \leftarrow$ the
      remainder keys in $S$.
   d. $(O.\downarrow_R).\downarrow_L \leftarrow NN$ ; $NN.\downarrow_R \leftarrow O.\downarrow_R$ ;
      $O.\downarrow_R \leftarrow NN$ ; $NN.\downarrow_L \leftarrow O$ .
   e. Join $k_1$ in $parent(O)$ and create a new pointer
      referring to $NN$.
   f. Increase $load(parent(O))$ by one.
   g. If $load(parent(O)) > M$,
      Invoke Split($parent(O)$, $k_1$).
3. If $O$ is a non-leaf node,
   a. $k \leftarrow$ the median of $S$.
   b. Allocate a new node $NN$ and the data larger
      than $k$ in $S$ are migrated to $NN$.
   c. $O \leftarrow$ the data smaller than $k$ in $S$.
   d. If $O$ is the root, create a new root $T$ containing
      $k$ and two pointers referring to $O$ and $NN$.
   e. Else
      i. Join $k$ in $parent(O)$ and create a new pointer
         referring to $NN$.
      ii. Increase $load(parent(O))$ by one.
      iii. If $load(parent(O)) > M$, invoke
         Split($parent(O)$, $k$).
```

The logic of algorithm **Split** is the same as that suggested by a traditional $B^+$-tree. Notably, splitting handles not only spend time overhead but also consume memory space. Algorithm **Delete** has also been modified to ensure high data aggregation around the arrived leaf. In general, $\delta = \lceil M/2 \rceil$ is a reasonable threshold used to determine when to activate data repartition or data merging between the target and the underflow siblings.

**Algorithm Delete**(*T*, *I*, $\delta$)
**Input**: The root *T* of a compressed $B^+$-tree and the data *I* to be removed.
**Output**: The root of the new compressed $B^+$-tree.

```
1. $N \leftarrow T$.
2. While $N$ is a non-leaf node,
     If $I \le N.k_1$,
       $N \leftarrow N.p_0$
     else for a specific key $N.k_s$ such
       that $N.k_s < I \le N.k_{s+1}$, $s \le load(N)$.
```

```
    $N \leftarrow N.p_s$.
3. If $I$ is not found, return 「Not found」 and exit.
4. Remove $I$ from $N$. Decrease $load(N)$ by one.
5. If $load(N) < \delta$, select one of the following cases:
  Case a: $N.\downarrow_R \neq null$ and $load(N) + load(N.\downarrow_R) \leq M$
      Merge the data in $N$ and $N.\downarrow_R$ into $N$. Remove the related key and pointer
      corresponding to $N.\downarrow_R$ at the parent. And then, execute data migration for
      achieving the smallest total data aggregation between $N.\downarrow_L$ and $N$.
  Case b: $N.\downarrow_L \neq null$ and $load(N.\downarrow_L) + load(N) \leq M$
      Execute the similar handles as Case a on the opposite direction.
  Case c: $load(N.\downarrow_L) + load(N) > M$ and
      $load(N) + load(N.\downarrow_R) > M$
      Evaluate and compare two measurements
      $\Delta(N.\downarrow_L) + \Delta((N)_R^-) + \Delta((N.\downarrow_R)_L^+)$ and
      $\Delta((N.\downarrow_L)_R^+) + \Delta((N)_L^-) + \Delta(N.\downarrow_R)$ to
      determine the direction for processing data migration and achieve the maximal data aggregation.
6. Return $T$.
```

## 5 Analytical Evaluations

The following system parameters are used to evaluate the storage requirement and query performance of a $CB^+$-tree.

- $B$: Memory block size;
- $t$: Integer size;
- $p$: Pointer size;

Every leaf node in a $CB^+$-tree contains many key values (entries) and two pointers. The order (capacity) of the external structure in a tree hierarchy is denoted by $M_1$. Then, a $CB^+$-tree's $M_1$ can be determined as following:

$$M_1 \times t + 2p = B \quad \Rightarrow M_1 = \left\lceil {(B-2p)}/{t} \right\rceil.$$

However, every leaf node in a traditional $B^+$-tree contains many key values and only one pointer. So, the $M_1$ of a traditional $B^+$-tree is $\left\lceil {(B-p)}/{t} \right\rceil$. In addition, entries in the non-leaf nodes of a $CB^+$-tree and a traditional $B^+$-tree both are pairs of a key value and a pointer to a sub-tree. The order of the internal structure ($M_2$) in a $CB^+$-tree and a traditional $B^+$-tree is smaller than $M_1$. As a result, $M_2$ is given as following:

$$M_2(t+p) + p = B \quad \Rightarrow M_2 = \left\lceil {(B-p)}/{(t+p)} \right\rceil.$$

We assume the total data amount for processing the indexing work is $N$. Since our $CB^+$-tree has full accommodation at the leaf level, the number of leaf nodes is determined as $\left\lceil N/M_1 \right\rceil$. For convenience, assuming that all nodes in the traditional $B^+$-tree are $\ln 2 \times 100\%$ (i.e. 69.3%) full on average [20]. Since $CB^+$-tree does not grantee the compact organization in its internal structure. The accommodation of non-leaf nodes in a $CB^+$-tree is also supposed to be 69.3% full. We account for the leaf and non-leaf nodes together, the total amount of nodes generated by a $CB^+$-tree is

$$\left\lceil \frac{N}{M_1} \right\rceil (1 + \frac{1}{M_2 \ln 2} + \cdots) \approx \left\lceil \frac{N}{M_1} \right\rceil \times (\frac{M_2 \ln 2}{M_2 \ln 2 - 1}).$$

On the other hand, the total amount of nodes generated by a traditional $B^+$-tree is

$$\left\lceil \frac{N}{M_1 \ln 2} \right\rceil (1 + \frac{1}{M_2 \ln 2} + \cdots) \approx \left\lceil \frac{N}{M_1 \ln 2} \right\rceil \times (\frac{M_2 \ln 2}{M_2 \ln 2 - 1}).$$

The height of a $CB^+$-tree and a traditional $B^+$-tree are approximated by $\left\lceil \log_{\lceil M_2 \ln 2 \rceil} \frac{N}{M_1} \right\rceil + 1$ and $\left\lceil \log_{\lceil M_2 \ln 2 \rceil} \frac{N}{M_1 \ln 2} \right\rceil + 1$, respectively. The data search in a $CB^+$-tree or a traditional $B^+$-tree is single path. To a certain extent, the query performance of a $CB^+$-tree and a traditional $B^+$-tree is quite similar. The time complexity is $O(\log_{M_2} \frac{N}{M_1})$.

In a general implementation, parameter $B$ is allocated as 4 Kbytes and parameters $t$ and $p$ employs 16 bytes and 32 bytes to represent the indexed data. For the completeness of analysis, we apply the data amount $N$=$10^6$ in the simulation. The related results are listed in Table 2. Notably, the number of leaf entries actually used in $B^+$-tree is $254 \times 69.3\% \approx 177$. The number of non-leaf entries actually used in $CB^+$-tree and $B^+$-tree is $85 \times 69.3\% \approx 59$. The ratio of storage requirement between $CB^+$-tree and $B^+$-tree is $\frac{4038}{5748} \approx 0.702$.

Table 2 The simulated results of $CB^+$-tree and $B^+$-tree when inserting $10^6$ data.

| Scheme | $M_1$ | $M_2$ | # leaves | # nodes | Height |
|---|---|---|---|---|---|
| $CB^+$-tree | 252 | 85 | 3969 | 4038 | 4 |
| $B^+$-tree | 254 | 85 | 5650 | 5748 | 4 |

## 6 Experimental Results on Spatial Data

Four spatial datasets: *SC*, *UK*, *CA*, and *LB* are employed to implement practical indexing applications. Dataset *SC* contains 36,176 points representing the coast line of Scandinavia, and dataset *UK* contains 39,828 points representing the boundary of the United Kingdom. Datasets *CA* and *LB* include 62,556 and 53,145 points representing locations in California and Long Beach County. They are given in Fig. 2.

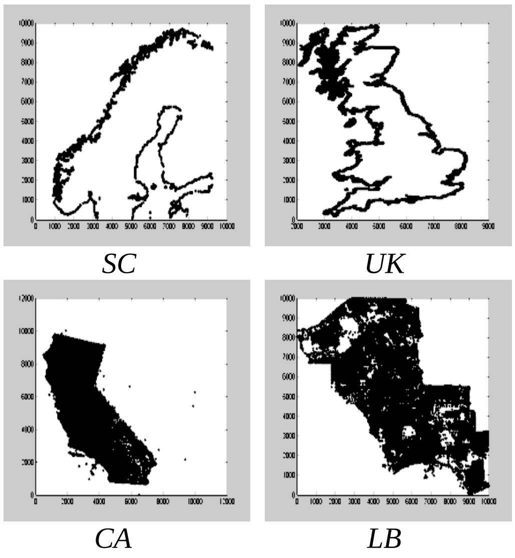
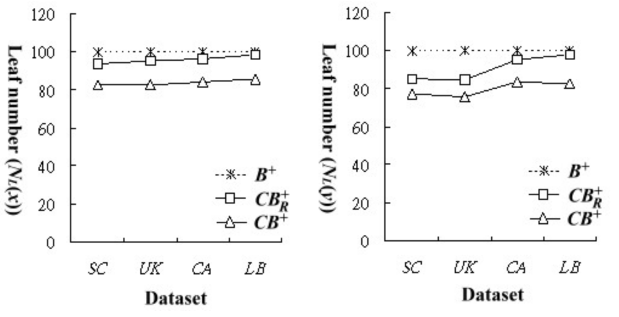


Fig. 2 Four spatial datasets.

Three index structures: conventional $B^+$-trees (abbreviated as $B^+$), $CB_R^+$-trees (abbreviated as $CB_R^+$), and $CB^+$-trees (abbreviated as $CB^+$) were implemented in C$^{++}$ programming language and executed on a workstation with an Intel Pentium 1.73 GHz processor. In this section, the experimental results contain three parts. The first part investigates the data behavior in external structures for all datasets. The second part proceeds to evaluate the entire structure behavior for all datasets. The third part measures the data aggregation for internal and external structure. For external structures, three measurements ($N_L$, $U_L$) were evaluated for comparative studies. $N_L$ is the total number of leaves generated in an index. $U_L$ is the average space utilization for all leaves. We note that the term $U_L$ multiplied by the node capacity $M$ can approximate the average number of data inserted between two consecutive splits. Namely, $U_L \times M$ indicates the splitting period. With respect to entire structures, four measurements ($N$, $S$, $P$, $U$) were investigated. $N$ is the total number of nodes generated in an index. $S$ counts the total number of splits in an index. $P$ and $U$ are the average splitting period and the total space utilization in an index.

For each dataset, the *x*- and *y*-coordinates of 2D geographical point data are handled respectively. Namely, all *x*-coordinate values in a dataset are indexed by $B^+$, $CB_R^+$, and $CB^+$. And then, all *y*-coordinate values are processed. All seven measurements as mentioned in section 6 are evaluated for analysis. Again, the maximum node capacity is fixed as 40 for all structures. $N_L(x)$ and $N_L(y)$ are the amounts of leaf nodes generated by indexing the *x*-coordinate and *y*-coordinate of spatial data. The results obtained from $B^+$ are taken as the benchmark and assigned as 100. As given in Fig. 3, $CB^+$ and $CB_R^+$ allocate the more economic amounts of leaf nodes than $B^+$. The data points in *SC* and *UK* appear to be a more sparse distribution than *CA* and *LB*. Using of traditional index schemes to deal with sparse data easily causes much vacant space in leaves. As learned from the behavior of $U_L(x)$ and $U_L(y)$ in Fig. 3, our method still keeps the space utilization of leaves at a high level no matter what the data distribution appears. The improvement of space efficiency for the external $CB^+$ varies from 15% to 25% as compared to $B^+$. Stated precisely, $CB^+$ utilize their external structures at the level from 82% to 87% and $CB_R^+$ at the level from 72% to 77%. While $B^+$ only performs from 64% to 71%.



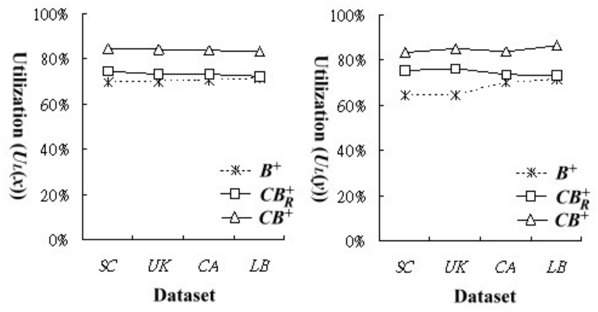


Fig. 3 The effect of spatial datasets on external index structures.

Fig. 4 presents the experimental results of total number of nodes ($N(x)$ and $N(y)$), total number of splits ($S(x)$ and $S(y)$), average splitting period ($P(x)$ and $P(y)$), and total space utilization ($U(x)$ and $U(y)$) for four datasets. Again, $B^+$ are taken as the benchmark structures when evaluating the behavior of total number of nodes and total number of splits. Since the number of leaves dominates the total number of nodes in an index; hence, $N(x)$ (and $N(y)$) exhibit a similar behavior to $N_L(x)$ (and $N_L(y)$). Besides, $N(x)$ and $N(y)$ also have a quite similar behavior to $S(x)$ and $S(y)$ due to the generation of non-leaf nodes is derived from leaf splitting. Fig. 4 summarizes these results, showing that compressed $B^+$-trees consume considerably fewer storage and time cost than $B^+$-trees.

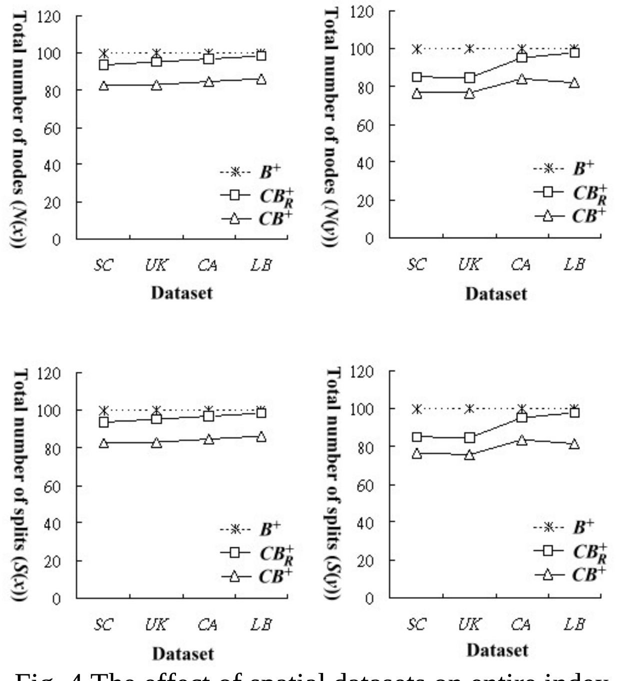


Fig. 4 The effect of spatial datasets on entire index structures.

An easy split has handles involving with one leaf node and the corresponding entry at the upper level. While a complicated split can propagate the splitting process upward to the root and involve several non-leaf nodes. In the building process of an index, splits are expensive and irreversible; hence, they should be invoked in the circumstance that no alternative handle can support the proper processing. Measurement $P$ in Fig. 5 reveal that $CB^+$ and $CB_R^+$ activate fewer splits and hence spend less time cost in completing their whole structures. As to the total space utilization $U$, the quite similar results to $U_L$ are shown in the last row of diagrams in Fig. 7.

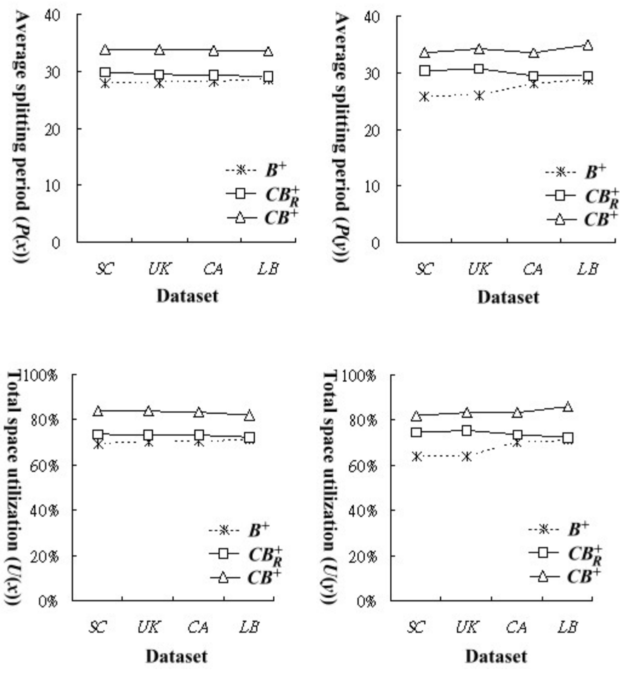


Fig. 5 The comparisons of time and space efficiencies.

Last but not least, the data aggregation $\Omega$ is investigated for all cases. As shown in the first row of diagrams in Fig. 6, $CB^+$ and $CB_R^+$ have the higher $\Omega$ values than $B^+$. The improvement appears apparently in the cases of $CB^+$ and sparse data distribution. Associating the experimental results of $U_L$ with $\Omega$, the data arrangement in leaves of $CB^+$ presents the most compact accommodation with the least data variance. However, although $CB^+$ and $CB_R^+$ generate the fewer interior entries than $B^+$, while these entries have diverse values for classifying lower data authentically. Namely, for achieving the high data aggregation at leaf level, $CB^+$ and $CB_R^+$ likely have the low data aggregation in their internal structures. Since our method does not deal with internal structures, compactness and aggregation are not improved in interior nodes. We term $\omega$ the data dispersion measured from the internal structure of an index, where $\omega = \sum_{i=1}^{\text{all non-leaves}} S^2(N_i)$. The $\omega$ values of $B^+$ are taken as the bench mark and the relative $\omega$ values of $CB^+$ and $CB_R^+$ are evaluated for comparison. As shown in the second row of diagrams in Fig. 6, no regular improvement of data dispersion is found.

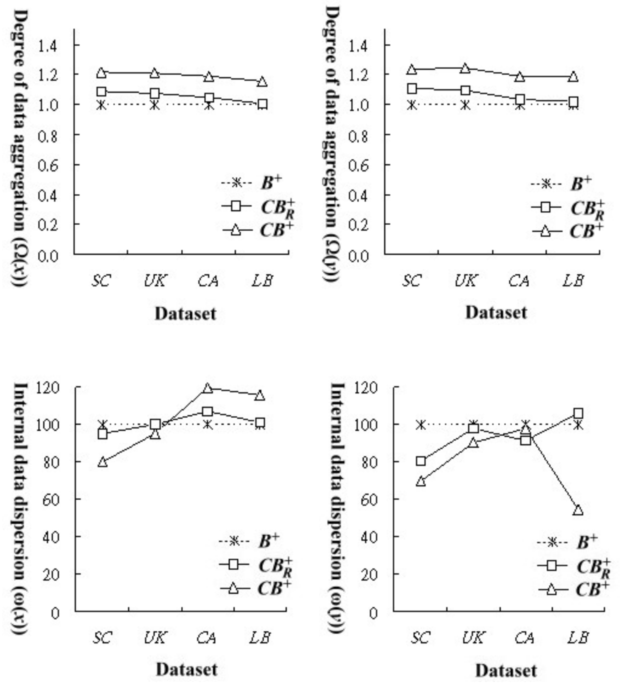


Fig. 6 Data aggregation and dispersion of external and internal structure.

## 7 Conclusion

In this paper, associating the utility of free space from the siblings around a full target leaf is suggested as a better policy than traditional node splitting policies. Three important contributions are provided by our index scheme. First of all, the dilemma derived from data insertion order is solved. Second, many unnecessary splits activated by traditional index schemes are eliminated completely. Third, the data classification becomes more authentic and reliable. The concrete efficacy brought to the practical database applications regarding to access performance is twofold. First, the main memory is fitted with the condensed index structures. Second, the caches are equipped with the higher quality of data and the less frequency of cache misses is caused. In addition, many mentioned techniques relating to computer architecture techniques or concurrency algorithm can collaborate with our method to achieve the even higher database performance.

*References:*

[1] A. Ailamaki, D.J. DeWitt, M.D. Hill, M. Skounakis, Weaving Relations for Cache Performance, *Proceedings of VLDB Conference*, *Roma*, *Italy*, 2001, pp. 169–180.

[2] A. Ailamaki, D.J. DeWitt, M.D. Hill, D.A. Wood, DBMSs on a modern processor: Where does time go? *Proceedings of VLDB Conference*, *Edinburgh*, *Scotland*, 1999, pp. 266–277.

[3] N. Beckmann, H.P. Kriegel, R. Schneider, B. Seeger: The R*-tree: an efficient and robust access method for points and rectangles, *Proceedings of ACM SIGMOD international conference on management of data*, *Atlantic City*, *New Jersey*, 1990, pp. 322–331.

[4] H. Berliner: The *B**-tree search algorithm: a best-first proof procedure, *Tech. Rep. CUM-CA-78-112*, *Computer Science Dept.*, *Carnegie-Mellon Univ., Pittsburgh*, 1978.

[5] P.A. Boncz, S. Manegold, M.L. Kersten, Database Architecture Optimized for the New Bottleneck: Memory Access, *Proceedings of VLDB Conference*, *Edinburgh*, *Scotland*, 1999, pp. 54–65.

[6] E. Chávez, G. Navarro, *A compact space decomposition for effective metric indexing, Pattern Recognition Letters*, Vol.26, No.9, 2005, pp. 1363–1376.

[7] S. Chen, P.B. Gibbons, T.C. Mowry, Improving Index Performance through Prefetching, *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, *Santa Barbara,* 2001, pp. 235–246.

[8] C.H. Goh, A. Lim, B.C. Ooi, K.L. Tan, Efficient Indexing of High-dimensional Data through Dimensionality Reduction, *Data and Knowledge Engineering*, Vol.32, 2000, pp. 115–130.

[9] R.A. Hankins, J.M. Patel, Effect of Node Size on the Performance of Cache-Conscious $B^+$-trees, *Proceedings of the 2003 ACM SIGMETRICS*, 2003, pp. 283–294.

[10] A. Henrich, H. S. Six, P. Widmayer, The LSD Tree: Spatial Access to Multidimensional Point and Non-Point Objects, *Proceeding of 15th VLDB Conference*, 1989, 45–53.

[11] H. Jagadish, B.C. Ooi, K.L. Tan, C. Yu, R. Zhang, iDistance: An Adaptive $B^+$-tree Based Indexing Method for Nearest Neighbor Search,

*ACM Transactions on Database Systems (TODS)* Vol.30, No.2, 2005, pp. 364–397.

[12] J. Kim, D.Y. Hur, J.S. Lee, J.S. Yoo, S.H. Lee, *Insertion method in a high-dimensional index structure for content-based image retrieval*, US Patent 6,389,424 2002.

[13] S. Lim, M.H. Kim, Restructuring the Concurrent $B^+$-tree with Non-blocked Search Operations, *Information Sciences*, Vol.147, No.1-4, 2002, pp. 123–142.

[14] K. Lin, H. Jagadish, C. Faloutsos, The TV-tree: An Indexing Structure for High-dimensional Data, *VLDB Journal*, Vol.3, 1995, pp. 517–542.

[15] J. Lukaszuk, R. Orlandic, On Accessing Data in High-Dimensional Spaces: A Comparative Study of Three Space Partitioning Strategies, *Journal of Systems and Software*, Vol.73, No.1 ,2004, pp. 147–157.

[16] R. Orlandic, B. Yu, A Retrieval Technique for High-dimensional Data and Partially Specified Queries, *Data and Knowledge Engineering*, Vol.42, No.1, 2002, pp. 1–21.

[17] J. Rao, K.A. Ross, Making $B^+$-Trees Cache Conscious in Main Memory, *Proceedings of ACM SIGMOD Conference*, 2000, pp. 475–486.

[18] B. Seeger, H.P. Kriegel, The Buddy-tree: An Efficient and Robust Access Method for Spatial Database Systems, *Proceeding of 16th VLDB Conference*, 1990, pp. 590–601.

[19] V. Srinivasan, M.J. Carey, Performance of $B^+$-tree Concurrency Control Algorithms, *VLDB Journal*, Vol.2, 1993, pp. 361–406.

[20] A. Yao, Random 2-3 trees, *Acta informatica*, Vol.2, No.9, 1978, pp. 159-170.

[21] B. Yu, T. Bailey, R. Orlandic, J. Somavaram, KDB$_{KD}$-Tree: A Compact KDB-tree Structure for Indexing Multidimensional Data, *Proceeding of International Conference on Information Technology, ITCC, Las Vegas*, 2003, pp. 676–680.

## Appendix

The totally ordered entries contained in node *A* and *B* are assumed to be $\{a_1, a_2, \cdots, a_n\}$ and $\{b_1, b_2, \cdots, b_m\}$. Their corresponding data variance $s_n^2(A)$ and $s_m^2(B)$ are given as following:

$$s_n^2(A) = \frac{1}{(n-1)}\sum_{i=1}^{n}(a_i - \overline{a_n})^2$$

$$s_m^2(B) = \frac{1}{(m-1)}\sum_{i=1}^{m}(b_i - \overline{b_n})^2 .$$

Term $A_R^-$ is used to denote the modified node of *A* whose rightmost entry ($a_n$) is removed. Then, $s_{n-1}^2(A_R^-)$ can be evaluated as following:

$$\begin{aligned}(n-1)s_n^2(A) &= \sum_{i=1}^{n}(a_i - \overline{a_n})^2 = \sum_{i=1}^{n-1}(a_i - \overline{a_n})^2 + (a_n - \overline{a_n})^2 \\ &= \sum_{i=1}^{n-1}(a_i - \overline{a_{n-1}} + \overline{a_{n-1}} - \overline{a_n})^2 + (a_n - \overline{a_n})^2 \\ &= \sum_{i=1}^{n-1}(a_i - \overline{a_{n-1}})^2 + \sum_{i=1}^{n-1}(\overline{a_{n-1}} - \overline{a_n})^2 + (a_n - \overline{a_n})^2 \\ &= (n-2)s_{n-1}^2(A_R^-) + (n-1)(\overline{a_{n-1}} - \overline{a_n})^2 + (a_n - \overline{a_n})^2\end{aligned}$$

So, $s_{n-1}^2(A_R^-)$ can be obtained by $\frac{1}{(n-2)}[(n-1)(s_n^2(A) - (\overline{a_{n-1}} - \overline{a_n})^2) - (a_n - \overline{a_n})^2]$. Term $A_L^-$ is the node modified by removing the leftmost entry of *A*. Then, $s_{n-1}^2(A_L^-)$ can be obtained by the following calculation:

$$\begin{aligned}(n-1)s_n^2(A) &= \sum_{i=1}^{n}(a_i - \overline{a_n})^2 = (a_1 - \overline{a_n})^2 + \sum_{i=2}^{n}(a_i - \overline{a_n})^2 \\ &= (a_1 - \overline{a_n})^2 + \sum_{i=2}^{n}(a_i - \overline{a_{n-1}} + \overline{a_{n-1}} - \overline{a_n})^2 \\ &= (a_1 - \overline{a_n})^2 + \sum_{i=2}^{n}(a_i - \overline{a_{n-1}})^2 + \sum_{i=2}^{n}(\overline{a_{n-1}} - \overline{a_n})^2 \\ &= (a_1 - \overline{a_n})^2 + (n-2)s_{n-1}^2(A_L^-) + (n-1)(\overline{a_{n-1}} - \overline{a_n})^2\end{aligned}$$

So,

$$s_{n-1}^2(A_L^-) = \frac{1}{(n-2)}[(n-1)(s_n^2(A) - (\overline{a_{n-1}} - \overline{a_n})^2) - (a_1 - \overline{a_n})^2].$$

Inversely, the term $B_R^+$ (or $B_L^+$) is used to denote the modified node of *B* which an entry is accommodated to *B*'s right side (or left side). Then, $s_{m+1}^2(B_R^+)$ and $s_{m+1}^2(B_L^+)$ are evaluated as following:

$$s_{m+1}^2(B_R^+) = \frac{1}{m}[(m-1)s_m^2(B) + m(\overline{b_m} - \overline{b_{m+1}})^2 + (b_{m+1} - \overline{b_{m+1}})^2]$$

$$s_{m+1}^2(B_L^+) = \frac{1}{m}[(b_1 - \overline{b_{m+1}})^2 + (m-1)s_m^2(B) + m(\overline{b_m} - \overline{b_{m+1}})^2]$$