Dynamic Hilbert Curve-based B^+-Tree to Manage Frequently Updated Data in Big Data Applications

Dongmin Seo, Sungho Shin, Youngmin Kim, Hanmin Jung, Sa-kwang Song

Dept. of Computer Intelligence Research, Korea Institute of Science and Technology Information, South Korea
{dmseo, maximus74, ymkim, jhm, esmallj}@kisti.re.kr

Abstract: In big data application sets, the values of the data used change continually in practice. Therefore, applications involving frequently updated data require index structures that can efficiently handle frequent update of data values. Several methods to index the values of frequently updated data have been proposed, and most of them are based on R-tree-like index structures. Research has been conducted to try to improve the update performance of R-trees, and focuses on query performance. Even though these efforts have resulted in improved update performance, the overhead involved and the immaturity of the concurrency control algorithms of R-trees render the proposed methods a less-than-ideal choice for frequently updated data. In this paper, we propose an update-efficient indexing method. The proposed index is based on the B^+-tree and the Hilbert curve. We present an advanced Hilbert curve that automatically adjusts the order of the Hilbert curve in sub-regions, according to the data distribution and the number of data items. We show through experiments that our strategy achieves a faster response time and higher throughput than competing strategies.

[Dongmin Seo, Sungho Shin, Youngmin Kim, Hanmin Jung, Sa-kwang Song. **Dynamic Hilbert Curve-based B^+-Tree to Manage Frequently Updated Data in Big Data Applications**. *Life Sci J* 2014;11(10):454-461] (ISSN:1097-8135). http://www.lifesciencesite.com. 62

Keywords: Dynamic Hilbert curve; B^+-tree, frequently updated data; multi-dimensional data; big data

1. Introduction

Big data has recently emerged as an important concept in information technology (IT). Big data poses a challenge for information technology in terms of providing effective counterplans and making more accurate predictions through the utilization and analysis of massive amounts of data. Governments and leading enterprises across the globe expect big data to become a major new source of economic value. Therefore, they are researching its effects on market trends, and studying new business models that use big data (Azhar et al., 2010). Data sets for big data applications are being collected at an increasing rate by ubiquitous information-sensing mobile devices, aerial sensory technologies (remote sensing), software logs, cameras, microphones, radio-frequency identification readers, and wireless sensor networks. The actual values of the data used in big data applications change continually. In the database community, much research has been conducted on frequent updation of data in an effort to meet the requirements posed by big data applications. A major issue in this area is the improvement of application performance by using efficient index structures for frequently updated data.

Generally, big data applications are characterized by a high volume of updates that occur when data values (i.e., multidimensional data) change frequently and continually. The high volume of updates poses several new challenges for the design index structures, one of which derives from the need to accommodate very frequent updates while allowing for the efficient processing of queries. This combination of desired functionalities is particularly troublesome in the context of indexing multidimensional data. The dominant indexing technique for multidimensional data with low dimensionality involves the use of R-tree families (Antonin, 1984). However, they have been conceived for largely static datasets and exhibit poor update performance. Several techniques based on R-tree families have been proposed for indexing frequently updated data, but while they are efficient for queries, their update performance is poor.

Consequently, a few techniques to improve the update performance of R-tree families have been proposed (Dongseop et al., 2002; Mongli et al., 2003). They employ lazy update and bottom-up update techniques to improve update performance. However, update performance remains an issue. The problem is exacerbated by the concurrency control algorithms of R-trees, such as the R^{link}-tree (Philip et al., 1981). These simply cannot adequately handle a high volume of concurrent access involving updates. Notably, frequent tree ascents caused by node splitting and the propagation of minimum-bounding rectangle (MBR) updates lead to costly lock conflicts. This problem is inherent in many multidimensional index structures. Another problem with existing index structures for frequently updated data is that they are not easily integrated into existing database systems.

Two studies in particular (Christian et al., 2004; Man et al., 2008) propose another approach, which uses classical B^+-tree indexing and is based on a space-filling curve to transform multidimensional data into one-dimensional values. There are several advantages to using the B^+-tree. First, the B^+-tree is used widely in commercial database systems and has proved to be very efficient with respect to queries as well as updates. It is both scalable and robust with respect to varying workloads. Second, being a one-dimensional index, it does not exhibit the update performance problems associated with R-tree families. Third, it is typically appropriate for modeling multidimensional data extents as points, which enables linearization and subsequent B^+-tree indexing. Usually, the order of the space-filling curve of an index is static. In practice, however, data are not uniformly distributed and the number of data points is not fixed. Thus, at a certain point in time, an excessive number of data points may have the same curve value. This situation can severely degrade the performance of indexing methods. Taking this shortcoming as our motivation, we propose a B^+-tree based on a dynamic Hilbert curve. This is a novel way of indexing frequently updated data based on the Hilbert curve, the order of which can be varied dynamically according to data distribution. Our proposed index structure dynamically adjusts the order of the space-filling curve according to the data distribution of sub-regions in order to mitigate performance degradation issues.

This paper is organized as follows: in Section 2, we review previous work directly related to ours. We present our proposed indexing method in Section 3, and show our experimental results in Section 4. Finally, we conclude this paper in Section 5 with directions for our future work.

2. Related Works

Traditional index structures for multidimensional data, such as R-trees and their variants (e.g., R^*-trees), were designed to support efficient query processing rather than to enable efficient updates. These index structures work well in applications where queries occur much more frequently than updates. However, they are not adequate for applications involving the indexing of frequently updated data because such applications bear workloads characterized by heavy loads of updates and frequent queries. Consequently, several new index structures have been proposed to index frequently updated data. The Lazy Update R-tree (Dongseop et al., 2002) aims to reduce update cost by handling separately updates of data that do not move outside their leaf-level MBRs. A generalized approach to bottom-up updates in R-trees has also been recently examined (Mongli et al., 2003). The Time Parameterized R-tree (TPR-tree) (Yufei et al., 2003) indexes linear functions of time. The current value of a data item is found by simply applying the function representing its value to the current time. MBRs are also functions of time. Specifically, in each dimension, the lower bound of an MBR is set to move with the maximum downward change of all enclosed data, while the upper bound is set to move with the maximum upward change of all enclosed data. As enclosed data can be both frequently updated points and rectangles, this approach ensures that the bounding rectangles are indeed bounding at all times. Frequent updates are needed to ensure that updated data currently in the vicinity are assigned to the same bounding rectangles. Further, bounding rectangles never shrink and are generally larger than needed. To counter this phenomenon, a so-called “tightening” is applied to bounding rectangles when they are accessed.

The aforementioned solutions cannot be easily integrated into an existing relational database, as considerable changes are required in the “kernel” of a system (e.g., query optimization, concurrency control, etc.). Jensen et al. (Christian et al., 2004) propose the B^x-tree, which consists of B^+-trees indexing the transformed one-dimensional values of multidimensional data based on a space-filling curve (e.g., Hilbert curve). In one study (Man et al., 2008), a B^{dual}-tree is proposed to enhance query efficiency by eliminating false hits of B^x-trees. These index structures employ the Hilbert curve to transform multidimensional data into one-dimensional data. In order to make this transformation, the order of the Hilbert curve must first be determined. The order influences the performance of the indexing methods. If a large order value is used for indexing, the index size becomes large. On the other hand, if we use a small order value, the number of data points whose curve values are identical increases. As previously mentioned, a large number of identical curve values decreases search performance.

3. Dynamic Hilbert Curve-based B^+-Tree

3.1 Naïve Approach based on the Hilbert Curve

In order to explain our index structure more clearly, we will first present the naïve approach followed by our proposed approach. The basic structure is similar to that of the B^{link}-tree (Srinivasan et al., 1993), the latter of which is a version of the B^+-tree that has been modified to support concurrency control techniques. Internal nodes at the same level in a B^{link}-tree are linked in concurrency control algorithms. The leaf nodes contain the Hilbert curve values of multidimensional data that are being indexed. The Hilbert curve is one of a variety of space-filling curves. A space-filling curve is a continuous path that visits every point in a discrete, multidimensional space exactly once and never intersects itself. Although other curves may be used, we use the Hilbert curve because it has been found to be slightly better than the Peano curve (Man et al., 2008; Bongki et al., 2001). A leaf entry of the B^{link}-tree is the (*cv*, *pid*) pair, where *cv* is a Hilbert curve value and *pid* is a data page identifier. Data pages contain information about multidimensional data. Our index decides the order of the Hilbert curve in favor of storing all data with the same *cv* value on one disk page. Consequently, the tree size is very small. It can also handle frequent updates of data, since regions covered by the *cv*s are virtually static unless the order of the Hilbert curve is changed.

It is very simple to construct our index using curve values, as shown in Figure 1. Before constructing our index, we acquire the Hilbert curve values of the data. We decide that the order of the Hilbert curve will be 2, according to the number of entries that can be stored on a disk page. Assume that this order is the optimal value for storage utilization. When the order is 2, the data space is divided into 16 sub-regions, and *cv*s are assigned to each sub-region. The leaf nodes of our index contain (*cv*, *pid*) pairs, and each leaf node has a maximum of two entries. Through our index, an inserter finds a data page for a new data item using its *cv*. As shown in Figure 1, data that have the same curve value are stored on the same data page. A data page can be represented by a *cv* value because the data with the same curve value are retrieved through *cv*.

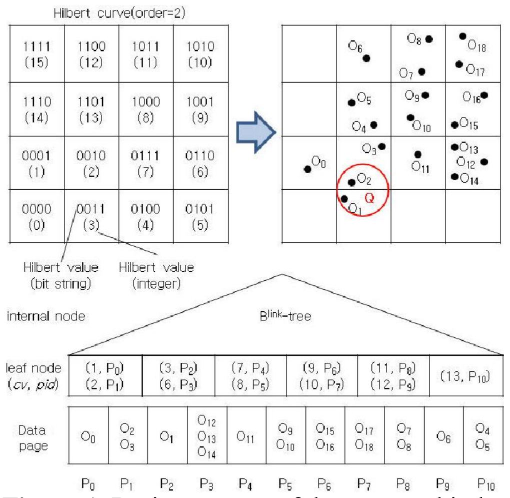


Figure 1. Basic structure of the proposed index

A range search is then performed, as follows. First, we acquire all *cv* values in the searcher's query range. For example, in Figure 1, the range of query *Q* overlaps with two regions that are covered by *cv*s 0010_b and 0011_b. We then traverse our index with *cv*s 0010_b and 0011_b, and locate leaf entries that contain P_1 and P_2. The searcher loads data pages P_1 and P_2 and then filters objects (O_1, O_2, O_3) stored in P_1 and P_2, in order to achieve the final results (O_1, O_2).

In this approach, index structures are very small because the order of the Hilbert curve is determined to maximize storage utilization. Furthermore, the index structures change much less frequently because regions covered by *cv*s are virtually static unless the order of the Hilbert curve is changed. However, in practice, data are not uniformly distributed at any given time. They can be highly skewed in sub-regions, and the number of data points in some data spaces may increase. Eventually, the number of data points with the same *cv* value exceeds the optimal level. For this reason, in spite of high storage utilization, the overall performance of the algorithm in terms of updating and searching is degraded. To solve this problem simply, we can use the highest order of the Hilbert curve. However, the performance of the algorithm will suffer because this lowers storage utilization. In particular, using the highest order severely affects update performance because it increases the likelihood of the curve value of an object being updated.

In Figure 2(a) and Figure 2(b), we assign Hilbert curve values to data when the order is 1 and 2, respectively. Both data spaces have the same size and O_1 and O_2 move along the same paths, which are denoted by arrows. In Figure 2, the curve value of O_1 does not change, but the O_2 curve value changes twice. Again, as the order increases, so too will the possibility of the curve values being updated. If the order of the Hilbert curve can be varied according to the data distribution and the number of data points in a certain sub-region, the performance of the index structures can improve. However, using different orders for different sub-regions makes mapping data to *cv*s impossible. In this paper, we propose a new indexing method based on the Hilbert curve that dynamically varies the order of the Hilbert curve in sub-regions. We also propose a technique to map the values of data to *cv*s in different curve orders.

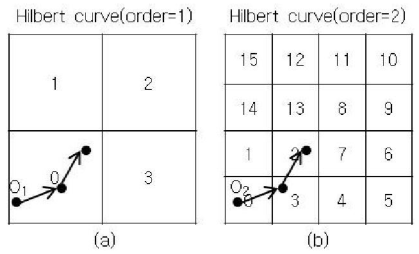


Figure 2. The order of the Hilbert curve and the likelihood of curve value updates

3.2 Construction Process

In Figure 3, we assume that the initial order of the Hilbert curve is 2 and the blocking factor of a data page is 4. Our index in the figure consists of a B^{link}-tree and data pages. A leaf entry of our index consists of (*key*, *pid*). A *pid* is the identifier of a data page where data having the same key are stored. A key is made up of (*o*, *cv*), where *o* is the order of the Hilbert curve and *cv* is the Hilbert curve value when the order is *o*. The order *o* is adopted dynamically according to the data distribution. An inserter performs insertion operations in three phases. In the first phase, the inserter creates a key for a newly inserted datum. In the second phase, the inserter locates the data page for the new key by traversing our index and adjusts the order of the Hilbert curve, if necessary. Finally, in the third phase, the new datum is inserted into the located data page. In this phase, if the object page cannot accommodate the new datum, the inserter performs split operations.

The initial order of the Hilbert curve is set by the user. With its location, the inserter calculates a *cv* value for the new datum. It then concatenates the bit string of *cv* and the bit string of the initial order *o* to create the key of the new datum. For example, in Figure 3(a), the *cv* of O_1 is 0001_b, and the order is 10_b. Thus, the key for O_1 is 100001_b (10_b + 0001_b). Even though both bit strings are concatenated, the key is not treated as an integer in the compare function of our index. We will show how to compare keys when we describe the next phase. In this phase, the inserter locates a proper data page for the new datum by traversing our index with the new key. As mentioned above, a leaf entry of our index consists of the key and the *pid*. If the inserter finds a leaf entry whose key is the same as the new key, it skips to the last phase. Otherwise, the inserter gets the *pid* of the previous leaf entry to make a new entry for the new datum, and places the new entry on the leaf page. In Figure 3(a), O_4 is the new datum to be inserted and there is no leaf entry that matches its key. The inserter thus fetches the *pid* of the previous entry to make a new entry and places the new entry (101010_b, P_1) on the leaf page. O_4 is then inserted into the *pid* data page in the next phase.

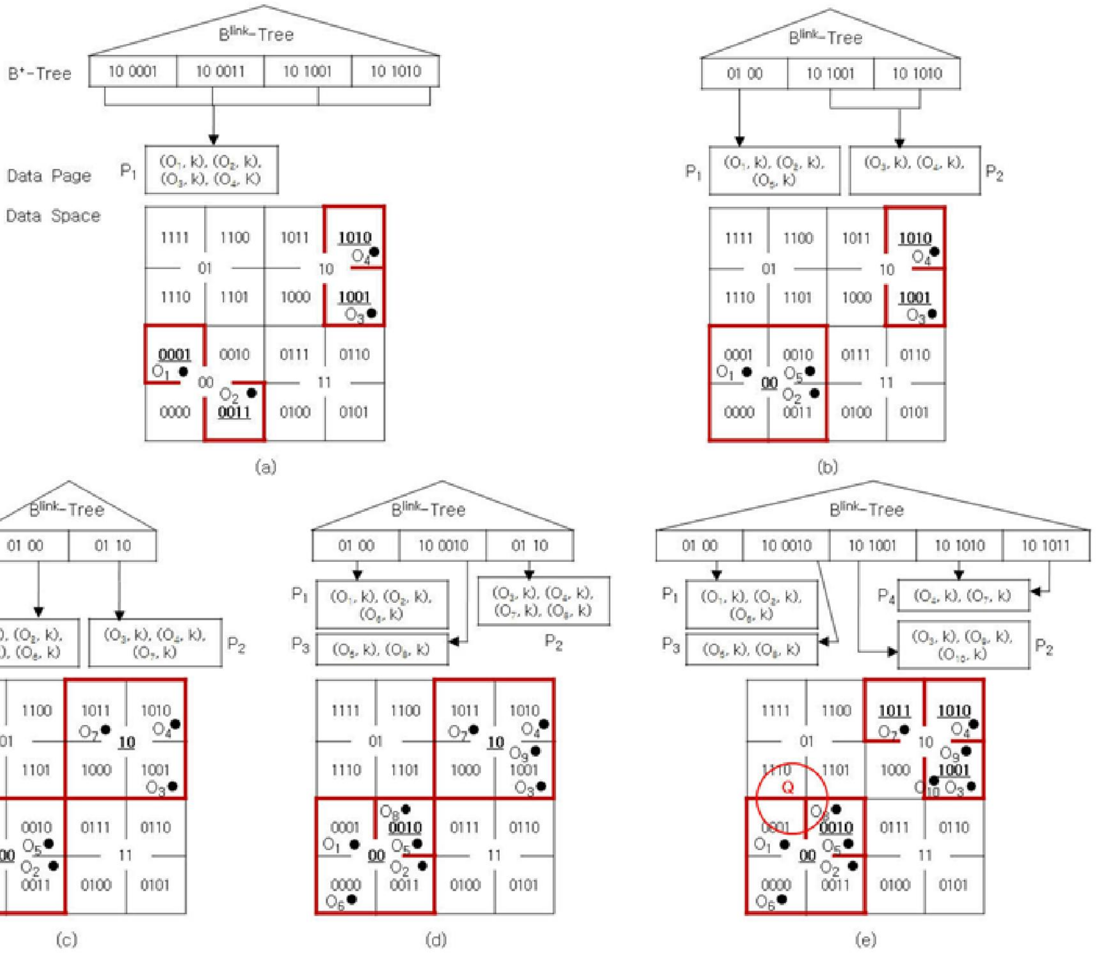


Figure 3. Construction processing phases of the proposed index

When locating a leaf entry in our index, the inserter examines neighboring keys and determines whether they can be merged with the new key. If the number of the neighboring keys that are covered with the lower-order key on the new key is 2, the new key is merged with those neighbors. However, if the number of neighboring keys is 1, the keys do not merge because this would increase dead space and degrade search performance. Merging the keys means that the sub-regions covered by them are merged as well, thus lowering their order. Assume that O_5 is being inserted into our index in Figure 3(a). The key for O_5, 100010_b, is placed between 100001_b and 100011_b, and the number of leaf entries in our index is 5. However, as previously mentioned, the inserter investigates neighboring keys and the new key to see if sub-regions covered by them can be merged to a lower-order sub-region. As shown in Figure 3(a), the sub-regions of O_1, O_2, and O_5 can be merged into sub-region 00_b. O_1, O_2, and O_5 are stored on the same page and are always loaded together. Thus, their curve values do not need to be assigned to each sub-region. In this case, we lower the order of the sub-region and assign the same key to the objects, i.e., O_1, O_2, and O_5. The existing keys (100001_b and 100011_b) are deleted, and a single new key for O_1, O_2, and O_5 namely, 0100_b is entered on the leaf node, as shown in Figure 3(b). Merging keys reduces the index structure without affecting search performance. Once the keys are merged, O_5 is inserted into data page P_1.

If the page has enough space, O_5 is saved on it. Otherwise, a split operation for the data page is initiated. The inserter divides the data on page P_1 and the new datum into two groups, according to the following grouping criteria: (1) data with a common key must be placed in the same group; (2) data with similar key values should be placed in the same group; (3) the number of data items in both groups should be similar; (4) each group should be filled to over 30% of its capacity. Since we assume that the maximum number of data items on a page is 4, O_5 cannot be placed in data page P_1. Consequently, the inserter starts a split operation. The sub-regions of O_1, O_2, and O_5 were earlier merged into the lower sub-region 00_b, so that the keys of P_1 and O_5 are ordered as 0100_b (O_1, O_2, O_5), 101001_b (O_3), and 101010_b (O_4). According to the above split criteria, the inserter stores O_1, O_2, and O_5 to P_1, and O_3 and O_4 to the new page P_2. Our index contains curve values derived from different orders, i.e., the order of 0100_b is 1, and the order of 1010001_b and 101010_b is 2. Keys in our index are arranged in order of ascending value of *cv*. If *cv*s of keys are the same, the order is decided based on the value of *o*. Therefore, our index uses a new key comparison method, as shown in Figure 4. The compare function compares key1 and key2 after converting the one belonging to a higher order to the order of the other. For example, assume that the new key 100000_b of O_6 is inserted into the tree in Figure 3(b). The inserter will compare the new key with the leaf entries of the tree to locate a data page for O_6. The new key 100000_b is transformed to 0100_b when compared with 0100_b on the tree. The new key is matched with 0100_b. Thus, O_6 is placed on data page P_1.

```
Function: Comparison
  Input: key1, key2
  Output: result (compResult, orderDiff)
Begin
  result.orderDiff = false;
  if ( key1.order < key2.order )
    key2.cv = transform key2.cv according to
    key1.order;
    result.orderDiff = true;
  else if ( key1.order > key2.order )
    key1.cv = transform key1.cv according to
    key2.order;
    result.orderDiff = true;
  end If
  result.compResult = key1.cv - key2.cv;
  return result;
End
```

Figure 4. Comparison function of the proposed index

Split keys may occur during the splitting of data pages. Assume that the new key 100010_b for O_8 is inserted into the tree in Figure 3(c). The data page P_1 will be located for O_8 by traversing our index. However, a data page can accommodate only four data items, and thus O_8 causes an overflow on page P_1. Since the data stored on P_1 have the same key, they cannot be divided into groups. The inserter thus splits keys. Splitting a key means that part of the sub-region covered by a key is assigned a higher order. If possible for a small index structure, the inserter tries to divide a sub-region into a higher-order sub-region and the original sub-region, as shown in Figure 5(a). Otherwise, the original sub-region is divided to four higher-order sub-regions, as shown in Figure 5(b). Keys are not created for sub-regions in Figure 5(b) that do not have any data. The key for the data on P_1 can be divided, as in Figure 5(a), since the sub-region covered by 100010_b contains two data items and the original sub-region contains three data items. Therefore, the inserter splits the key for the sub-region covered by 0100_b into 0100_b and 100010_b, as shown in Figure 3(d). In this case, traverses for insert or search operations must be carefully conducted when comparing keys. In Figure 3(d), assume that the inserter is trying to insert a new datum with key 100010_b in the tree. After comparing 100010_b with

0100_b, the inserter stops to traverse the tree, since the inserted key is covered by 0100_b, and tries to enter the new data item on page P_1. However, the new datum must be placed in P_3 because the leaf entry (100010_b, P_3) is on the tree. To find the same key or the covering key with the highest order among the inserted keys, the inserter must confirm the next keys of the key covering the inserted key on visited internal node or leaf node until the inserter meets the same key or the larger key than the inserted key. This search operation is not very costly, since entries of visited internal nodes or leaf nodes are in the main memory, where this operation is performed.

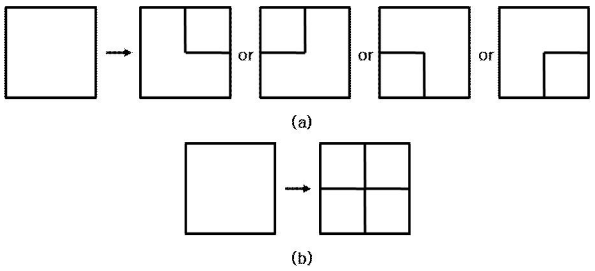


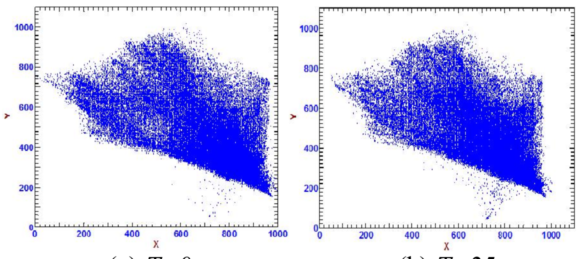
Figure 5. Key split strategies of the proposed index

3.3 Range Search

A range search is performed in a simple manner. As shown in Figure 3(e), in order to process a range query *Q*, we find all keys covered by regions that overlap with *Q*. Search keys consist of the maximum order and the curve values of overlapped regions. In the figure, the search keys of *Q* are 100001_b, 100010_b, 101110_b and 101101_b. We then find through our index *pid*s whose keys match the search keys. The searcher, like the inserter, tries to find the same key or a covering key with the highest order on the searched key. In the figure, we will find 0100_b and 100010_b.

4. Performance Evaluation

In this section, we experimentally compare our index against the B^{dual}-tree and the TPR-tree. All experiments were performed on a machine with a Pentium IV 3GHz CPU and 1GB of memory. The disk page size was fixed at 1KB. All reported I/O costs correspond to page accesses. We generated spatiotemporal data following the methodology of (Man et al., 2008; Yufei et al., 2003). The data space was two-dimensional, where each dimension had a domain of [0, 1000]. Five thousand rectangles were sampled from a 128,000 (128-K) spatial dataset (Man et al., 2008). It models the positions of airports. Each object is an aircraft that moves along a line segment connecting two airports. Initially, each aircraft is positioned at an arbitrary airport and randomly selects another airport as its destination. At subsequent timestamps, the aircraft will move from the source airport to a target one with a speed that is generated in the range [0, 5]. As soon as the object reaches its destination, it chooses another airport as its next destination, with a new speed obtained in the same way as before. In addition to these updates, an aircraft also issues an update 25 timestamps (= *T*) after the previous one (Man et al., 2008; Yufei et al., 2003). An index with a time horizon of $H = 2T = 50$ time units is created for each dataset. All objects are created and inserted into the index at time 0. Queries are issued after *H/2*. We measure the query cost by averaging it over a workload of 100 queries. Range queries are generated with a range of 0.01%, 0.02%, 0.03%, 0.04% and 0.05% of the total data space, and are uniformly distributed. *k*NN queries were generated with *k* at 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 150, 200, 500 and 1000, and are uniformly distributed. Figure 6 shows positions of the aircraft at $T = 0$ and $T = 25$.



(a) $T=0$ (b) $T=25$

Figure 6. Airplane positions

We compared the average update cost of our index against those of the others. Note that for each update, one deletion and one insertion were performed. We calculated the average update cost, including the deletion and insert costs. Figure 7 shows the average update cost of three indices with respect to dataset size. We recorded the number of I/Os whenever 5,000 (5K) updates were performed and averaged the total I/Os. As shown in the figure, our index and the B^{dual}-tree achieved significant improvement over the TPR-tree. This is on account of the properties of the B-tree and R-tree families. Usually, when the storage requirements of both index structures are similar, the insert and delete costs of R-tree families are higher than those of B-tree families. It seems from Figure 7 that our index only slightly outperforms the B^{dual}-tree. However, our index actually performs 1.5~2 times better than the B^{dual}-tree. The reason for this is similar to that for the storage utilization improvement. Our index does not need to keep a high order of the Hilbert curve. It flexibly adjusts the order of lower values according to the data distribution and the number of objects without loss of performance. Figure 8 shows the update cost with respect to the number of updates. Three index structures are constructed using 100,000 (100K) objects. This experiment has aspects similar to those in Figure 7. Our index still performs approximately four times and two times better than the TPR-tree and B^{dual}-tree, respectively.

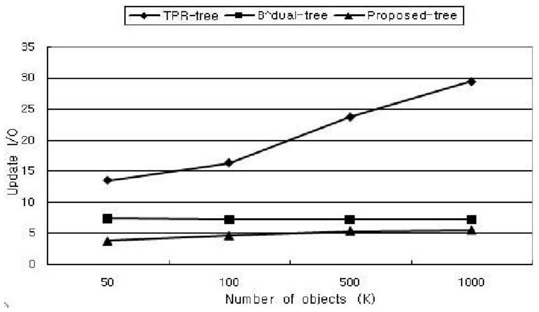


Figure 7. Update costs with varying data size

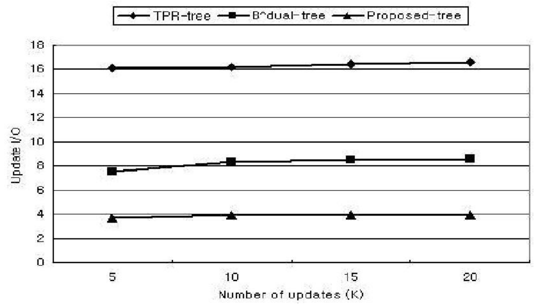


Figure 8. Update costs when the number of updates is varied

Figures 9 and 10 show the range of query performance of the three index structures. Figure 9 shows the query performance of the three index structures with respect to dataset size. We recorded the numbers of I/Os of the three index structures whenever 100 range queries were performed. The query size is 0.03%. The number of I/Os increases linearly with an increase in the number of objects. Our index outperformed other indices as the number of objects increased. The performance gap also increased.

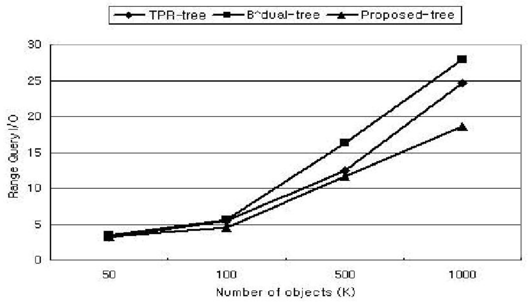


Figure 9. Range query costs with varying the data size

Figure 10 shows query performance while varying query size. The three index structures were constructed with 100,000 (100K) moving objects. While the number of I/Os of the TPR-tree and B^{dual}-tree increased linearly, our index maintained a constant number of I/Os.

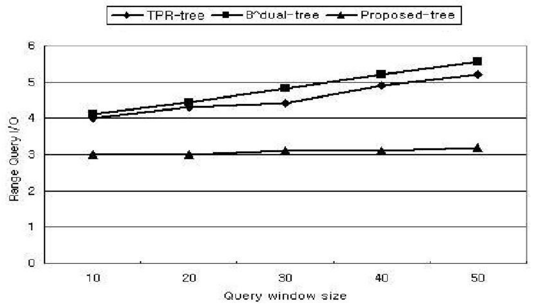


Figure 10. Range query costs with varying the range of query size

Figure 11 shows the target positions of the experimental *kNN* queries with *k = 5*. Figure 12 shows the *kNN* query performance of our index and the B^{dual}-tree with respect to the 100K dataset described in the experimental setup. The results in Figure 12 show that the performance of the *kNN* query for our index outperformed the B^{dual}-tree.

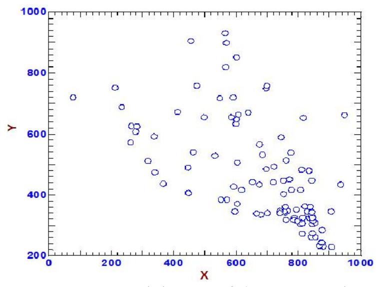


Figure 11. Target positions of *kNN* queries with *k*=5

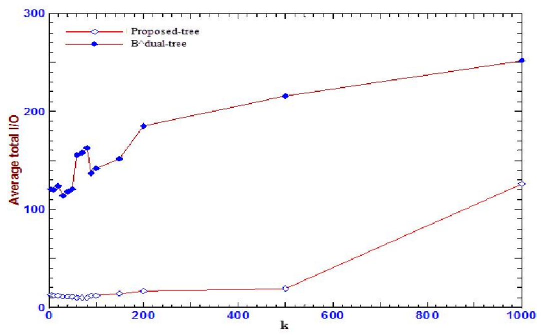


Figure 12. *kNN* query costs with varying the number of *k*

This is because the formula used by the *kNN* query algorithm in the B^{dual}-tree to compute searching radius increment yields a small value. This causes the *kNN* query algorithm of the B^{dual}-tree to slowly expand the searching area. The results also show that our index structure is superior to the B^{dual}-tree structure when handling non-uniform datasets. This is because our index dynamically adjusts the Hilbert curve order in crowded areas based solely on the input dataset, whereas the B^{dual}-tree uses a static set of reference points. Since our index divides the space based entirely on the input dataset, it always performs well with any dataset. However, for the B^{dual}-tree, if the predefined static reference points do not fit well with the input dataset, it will not perform well. If we look at the input dataset and calibrate the reference points in the B^{dual}-tree before performing the experiments, it will yield better results. It is a disadvantage of the B^{dual}-tree that information about the dataset is needed in order to tune the reference points to fit it. In both index structures, the number of I/Os increases linearly with the value of k. When $k < 500$, the number of I/Os of our index increases by a small value as k increases. This is because of the non-uniform dataset used. In crowded areas, the *kNN* query does not need to search a large area to obtain fewer than 500 objects.

5. Conclusions

In this paper, we proposed a new indexing method based on the Hilbert curve technique for moving objects. The contributions of our proposed method can be summarized as follows: first, we proposed a dynamic Hilbert curve technique that adjusts the order of a certain sub-region according to the data distribution or the number of objects. This technique reduces the index size. As a result, the performance of the update and query operations improves significantly. Second, the base data structure of our indexing method is the B^{+}-tree, and thus, it can easily be integrated into commercial database management systems. Finally, through experiments, we showed that our indexing method outperforms the TPR-tree and the latest B^{dual}-tree indexing method. In future work, we will apply our indexing method to various big data applications and experimentally compare performances of the applications.

Acknowledgement

This work was supported by the IT R&D program of MSIP/KEIT. [2014-044-024-002, Developing On-line Open Platform to Provide Local-business Strategy Analysis and User-targeting Visual Advertisement Materials for Micro-enterprise Managers].

Corresponding Author:
Dr. Sa-kwang Song
Department of Computer Intelligence Research
Korea Institute of Science and Technology Information
Daejeon, 305-806, South Korea
E-mail: esmallj@kisti.re.kr

References

1. Azhar R, Adnan A, Saeed M, Shah K. The performance of mapreduce over the varying nature of data. J. Life Science 2013;10(4):1263-1266.
2. Antonin G. R-trees: a dynamic index structure for spatial searching. Proc. Management of Data 1984;14(2):47-57.
3. Dongseop K, Sangjun L, Sukho L. Indexing the current positions of moving objects using the lazy update R-tree. Proc. Mobile Data Management 2002;113-120.
4. Mongli L, Wynne H, Christian S.J, Bin C, Kenglik T. Supporting frequent updates in R-trees: a bottom-up approach. Proc. Very Large Database 2003;29:608-619.
5. Philip L.L, Bing Y. Efficient locking for concurrent operations on B-tree. J. ACM Transactions on Database Systems 1981;6(4):650-670.
6. Christian S.J, Dan L, Beng C.O. Query and update efficient B^{+}-tree based indexing of moving objects. Proc. Very Large Database 2004;30:768-779.
7. Man L.Y, Yufei T, Nikos M. The B^{dual}-tree: indexing moving objects by space filling curves in the dual space. J. Very Large Database 2008;17(3):379-400.
8. Yufei T, Dimitris P, Jimeng S. The TPR^{*}-tree: an optimized spatio-temporal access method for predictive queries. Proc. Very Large Database 2003;29:790-801.
9. Srinivasan V, Michael J.C. Performance of B^{+}tree concurrency control algorithms. J. Very Large Database 1993;2(4):361-406.
10. Bongki M, Jagadish H.V, Christos F, Joel H.S. Analysis of the clustering properties of Hilbert space-filling curve. J. IEEE 2001;13(1):124-141.