Perfect Hashing Base R-tree for Multiple Queries

Parth Patel
Computer Science & Engineering Department
Thapar University
Patiala, India

Dr. Deepak Garg
Computer Science & Engineering Department
Thapar University
Patiala, India
dgarg@thapar.edu

Abstract—***Indexing of data is important for the fast query response in the information retrieval. Support of multiple query on the multidimensional data is a challenging task. Indexing of multidimensional data received much attention recently. In this paper a new data structure Perfect Hash Base R-tree (PHR-tree) is proposed. Node of PHR-tree is expansion of traditional R-tree node with Perfect Hashing Index to support multiple queries efficiently. It supports point query on the multidimensional data efficiently. It provides space efficiency and fast response to query (O(log n)) on all type of queries.***

Keywords—R-tree, Perfect Hashing, Range Query, Point Query.

I. INTRODUCTION

Point Query determines whether given item is in the data set or not and Range Query finds all the possible items whose range value is in the query. Performance of point query and range query depends heavily on the reliable, scalable and efficient system. In order to improve the performance of the query and scalability of the system, many structures are studied, e.g. Hash-based distributed structures like hash table [2], many tree structure like and variants of tree structures were studied and implemented like Compact B-tree, B+-tree, UB-tree, 2-3-tree etc. This type of single dimensional data structures can only support exact matching point query. Some efforts done for fast point query, e.g. Group-Hierarchical Bloom filter Array (G-HBA) and RBF [5], but they failed to provide multiple-query services. When we use bloom filter, inaccurate query results may be returned due to false positives. A minimal perfect hash function is a function that maps a set of n keys into a set of n integer numbers without collisions, so no false positives, but fails to support the range query. Single dimensional data structures are space inefficient and their query result depend upon the attribute value and attribute identifier storage structure. Data structures for the multidimensional data like R-tree, R+-tree, R*-tree, Hilbert R-tree, Priority R-tree etc. are studied over the years. Research is going on to facilitate data storage, its management and manipulation. Although R-tree structure used for multidimensional data supports range query quiet efficiently, it cannot support point query efficiently because R-tree only maintains the bounding boxes of multidimensional data and the pointers to actual data.

To get the point query, result needs to store the item identities in the leaf node, but this requires large storage space when used for the real world data. Perfect hashing is an efficient design for point query but it do not support range query, cover query. It uses hashing and do not store multidimensional range information of items. This paper presents a perfect hash base R-tree (PHR-tree), which integrates Perfect Hashing into R-tree nodes. The PHR-tree is essentially an R-tree structure to support dynamic indexing, in which each node maintains a range index, in order to indicate the attribute range of existing items. PHR-tree takes advantage of fact that range and cover queries are related to the item viewed as the answer to a range query. This mean that both range query and cover query can be supported in a single unified structure that stores both the items and the ranges of their attributes together. In addition, point query call similar operations too. Thus in PHR-tree, the range query and cover query are supported following the branch of R-tree while point query is mostly served in the branch of perfect hashing.

II. RELATED WORK

In this section, we briefly describe previous work in three areas relevant to the proposed new data structure PHR-tree: Perfect Hashing, R-tree and related tree structures supporting distinct queries. In order to achieve deterministic lookup times, a perfect hashing scheme, as shown by Kumar et al. in [2], is very effective. Their work proposes the adoption of a small fast table of discriminator values which, together with the key, are fed to a regular hash function, thus removing collisions and achieving perfect hashing. The R-tree structure [12] can efficiently support range query by maintaining index records in its leaf nodes containing pointers to their data. The completely dynamic index structure is able to provide efficient query service by visiting only a small amount of nodes in a spatial search. The index structure is height balanced. The path length from the root to any leaf node is identical, which is called the R-tree height. In essence, the family of R-tree index structures, including R+-tree [4][10] and R*-tree [5][11], uses solid Minimum Bounding Rectangles (MBRs), i.e., bounding boxes, to indicate the queried regions. The MBR in each dimension denotes an interval of the enclosed data with a lower and an upper bound. A lot of work which aims to

978-1-4799-2572-8/14/$31.00

support range query efficiently has been done. In essence, existing index structures for range query often hierarchically divide data space into smaller subspaces, such that the higher level data subspace contains the lower level subspaces and acts as a guide in the range query. Such work, however, cannot efficiently support both range query and point query. Some existing work may have similar design purpose with our PHR-tree, e.g., supporting two distinct queries in a unified structure. PHR-tree, however, enhances query functions to efficiently support four types of queries for items with multiple attributes in O(log n) time complexity. Moreover, our proposed PHR-tree utilizes space efficient storage design and deviates internal nodes routing (i.e., hash result probing on the same positions), providing fast response to user queries. One of the benefits using tree-based structures is to efficiently support range based queries, such as range query and cover query, which cannot be supported by conventional hash-based schemes. VBI-tree [6] provides point and range query services and supports multiple index methods in a peer-to-peer network, which, however, is unable to support bound query. BATON [7], a balanced binary tree, can support both exact match and range queries in O(log n)steps in a network with n nodes. It requires certain messages to provide load balance and fault tolerance. Distributed segment tree (DST) [13] focuses on the structural consistency between range query and cover query. It needs to, respectively, insert keys and segments to support these two queries. SD-Rtree [3] intends to support point and window (range) queries over large spatial datasets distributed at interconnected servers by using a distributed balanced binary spatial tree. In addition, the main difference between BR-tree and RBF [5] is that the latter only hashes the content of the root into its correlated Bloom filter which is then replicated to other servers. Though RBF can achieve significant space savings, it cannot provide exact-matching services or support complex queries in a distributed environment.

III. PHR-TREE STRUCTURE

In this section, we first briefly describe the basic architecture of the proposed PHR-tree. We then analyze load balance in the proposed PHR-tree and finally use a simple example to explain possible multiple queries.

A. Proposed Architecture

A PHR-tree is composed of root, internal, and leaf nodes. Fig. 1 shows an example of the proposed PHR-tree structure. A PHR-tree node combines an R-tree node with an extra Perfect Hash Index where a Perfect Hash Table is an nm-bit array representing a set of n items after applying perfect hash function on the item set. Because an R-tree node can exhibit a series of multidimensional attribute ranges and a Perfect Hash can display items in those ranges, the combined structure encompasses multidimensional ranges to cover an item's attributes (e.g., p attributes) in the R-tree node and stores the hashed value of an item identifier in the Perfect Hash Index.

In PHR-tree structure we use the Dynamic Perfect Hash Function for the Perfect Hash Index. A node in the Index is created whenever new value is inserted and node is deleted along with deletion of the item. Perfect Hash Function given below is Dynamic Perfect Hash Function which has set of Hash function and two level hash table for the Perfect Hashing Index. Set H of Hash function

h : U0; 1…m1 is called perfect if

$$|hH : h(x) = h(y)| = |H|/m \text{ for all } x,yU, x \neq y \quad (1)$$

then the Hash Function

$$h_a = \sum_{i=0}^{r} a_i k_i \, mod m \quad (2)$$

Size of the second level table is m^2_i, that is equal to

$$\left(\frac{m_i}{2}\right)\frac{1}{m_i^2} < \frac{1}{2} \quad (3)$$

Expected size of all tables are sum of first level and second level tables, this sum equals to

$$n + 2E\left[\sum_{i=0}^{n-1}\left(\frac{m_i}{2}\right)\right] = n + \frac{2\left(\frac{n}{2}\right)}{n} < 2n \quad (4)$$

So, expected space is less than 2n and we have constant probability of success each time.

B. Comparison of PHR-tree structure and Other State-of-Art-Structures

PHR-tree is different from other state-of-the-art structures, including Bloom filter, baseline R-tree, BATON [7], VBI-tree [6], DST [13], SD-Rtree [3],and RBF [5]. PHR-tree can achieve comprehensive advantages. PHR-tree has a bounded O(log n) worst case complexity for point query. The Perfect Hash Index in the root of PHR-tree can provide fast query result with O(1) complexity. In applications requiring exact query results, we can follow the Perfect Hash Index branch of PHR-tree to a leaf node to verify the presence of the queried item, with the searching complexity of O(log n). In such O(log n) complexity for point query, the real query latency is very small. The Bloom filter in the root of BR-tree can provide fast query result with O(1) complexity which is same as PHR-tree structure. However, the result may not be accurate due to false positive. Since Perfect Hashing have the same number of hash functions and counters, we need to carry out the hash-based computations for a queried item only once. The queried item checking on Perfect Hash Index can directly probe the same counters, saving much query time. Because we mainly follow the R-tree part in PHR-tree to obtain range and cover query services, these queries have the same complexity as R-tree to be O(log n). Meanwhile, PHR-tree structure can support bound query by checking the Perfect Hash Index along query path from the root to a leaf node, achieving O(logn) complexity. Perfect Hashing Index structure is an efficient design, which is also adopted in the PHR-tree.

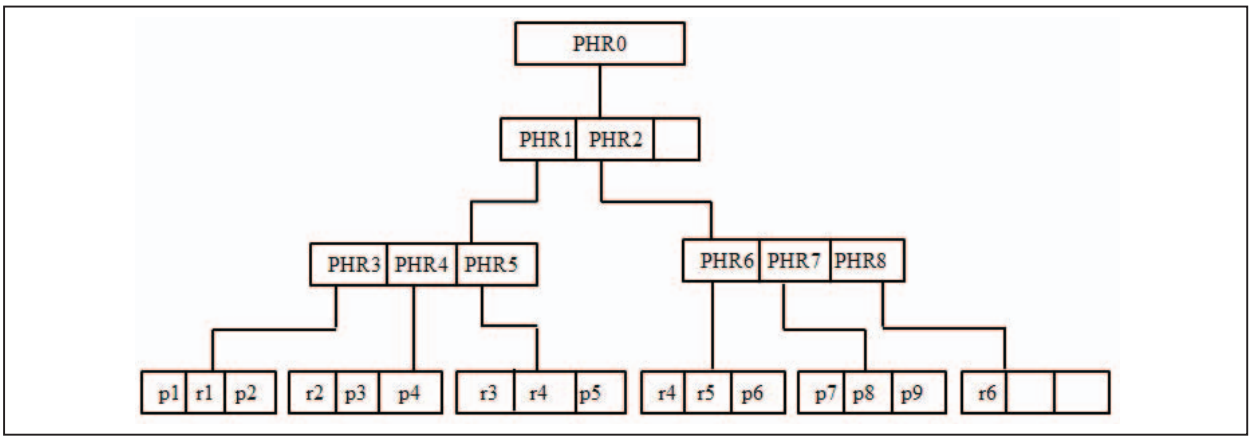

Fig. 1. PHR-tree Example

BATON and VBI-tree are mainly providing virtual indexing frameworks. Their practical performance mainly depends on the underlying used structures, not on themselves.

Although RBF support point query, its query result is not exact matching. Normally, the baseline R-tree cannot support point query. However, it can do so if we specially concatenate multidimensional attributes of an item as its identity. As a result, we can compare our PHR-tree with baseline R-tree in terms of point query, with SD-R tree in terms of point and range queries, and DST in terms of range and cover queries.

C. Example of Multiple Queries

Fig. 1 exhibits an example of PHR-tree structure. Our current data set, represented as a PHR-tree with root node PHR0, contains two subsets, PHR1 andPHR2, respectively, having subsets, PHR3, PHR4, PHR5 and PHR6, PHR7, PHR8. We store data objects (represented as points p) and ranges (represented as ranges r) into our PHR-tree structure. Fig. 2 explicitly describes multiple operations, including point, range, cover, and bound queries, for items with two attributes, i.e., (x, y), in a two-dimensional space. The operations of point query using PHR-tree become very simple and can be fast implemented compared with previous R-tree structures. For example, if we need to know whether item point belongs to our data set, we need to check the Perfect Hash Index along the query path from node PHR0, PHR2, to PHR7 by computing the hash values of item point. Perfect Hash Index will return positive to the existence of point item in this example. Given the outside point in Fig. 2, the Perfect Hash Function will return negative after the computation of hash functions for item point. Note that the point query in BR-tree actually can be executed with the complexity of O(1) only in the root that will cause a small false positive originated from Bloom filters, but in case of perfect hashing no false positives. Here, the result is always positive, if the item is in the Index. The processing of a range query starts from the root. If there is a node entry whose MBR intersects the query region, its sub tree is recursively explored. When the range query encounters a leaf node, we get all items whose bounding rectangles intersect the query region.

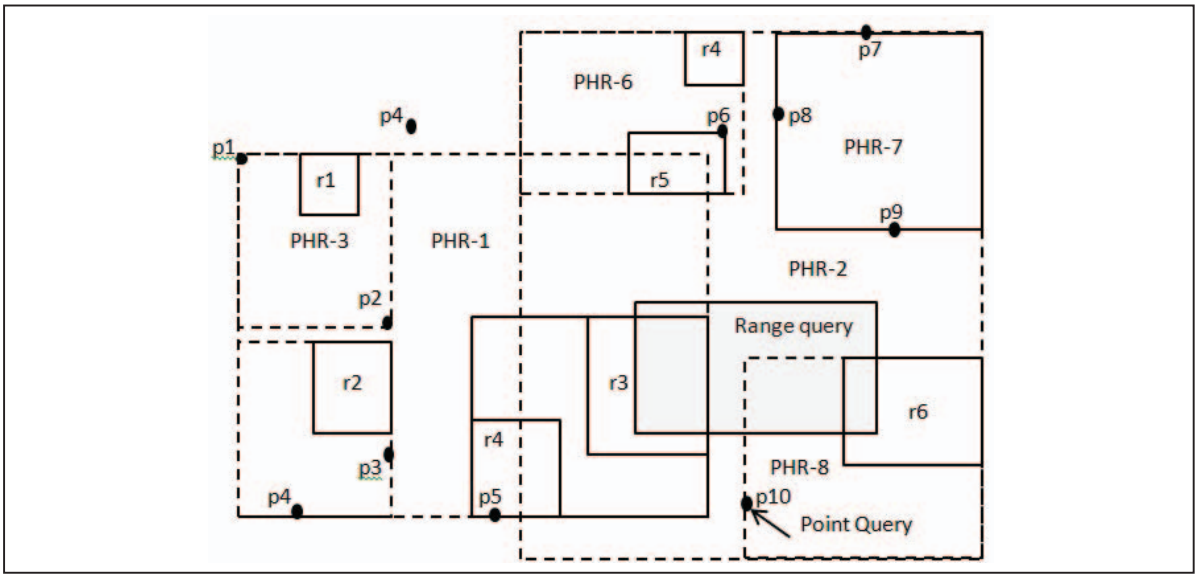

Fig. 2. Example of Multiple Queries on PHR-tree

For example, the shaded region in Fig. 2 intersects MBRs of both leaf nodes, PHR3 and PHR5. Items points will be returned for the range query as a result. A cover query is to obtain all multidimensional ranges. Cover a given item, For example, given an item X in Fig. 2, a cover query can determine that the two-dimensional bounding ranges r3 and r5 can cover it after query operations along the path from PHR0, PHR2 to PHR6 and PHR7 that contain r3 and r5.The operations of bound query are similar to those of point query. Given an item represented as a point, we need to check Perfect Hash Index along the query path from the root to a leaf node. When a leaf node containing the queried item is found, the multidimensional ranges linked to the leaf node are the queried bounds. For example, given an existed item p3 in Fig. 2, we know that p3 is contained in the leaf node PHR4. Thus, the shaded area, i.e., PHR4, denotes the multidimensional bounds on item p3 that will be the bound query result. In this way, we can quickly obtain approximate multidimensional attribute ranges of an item without querying its explicit attributes. In practice, the space-efficient index structure of PHR-tree can be fully deployed into high-speed memory to provide fast query services. Although we can get tighter bounds of items for bound queries by setting tighter MBRs on leaf nodes, the PHR-tree depth will become larger and more storage space will be required.

IV. LOCAL OPERATIONS

A. Item Insertion

```
Insert(Data d, PHR-tree)
Leaf Node = Choose_leaf(Data D, PHR-tree);
if Count(leaf Node) R then
        Leaf Node = Quad_Split(Leaf Node);
end if
Insert(Data d, Leaf Node);
Current Node = Leaf Node;
While (Current node NULL) do
        If(D not belongs to MBR (Current Node)) then
                ExMBR(Current Node)
        end if
         Insert(Data D, Perfect Hashing(Current Node));
        Current Node = Parent Node(Current Node);
end while
```

Insertion of an item into a PHR-tree includes operations on the R-tree and corresponding Perfect Hashing Index. Since an inserted item needs to be placed in a leaf node, we need to first find the leaf node and then insert it. We use CurNode to denote a currently checked PHR-tree node. The suitable leaf node for the item can be found in O (log n) time, by examining a single path. After locating the leaf node for the new item, we can carry out node insertion. If the leaf node has room for the new item, i.e., the number of entries is less than R, we can execute direct insertion operations by adding item pointer into the leaf node, hashing the item into Perfect Hash in the leaf node and all its parents node till the root. This process is in O(logn) time complexity. Otherwise, we need to split the leaf node by use of quadratic-cost algorithm, into two leaf nodes, i.e., the old one containing old entries and the new one containing item a. The insertion algorithm can be applied to insert a point or a range object, while taking its identity as the input to an associated Perfect Hash Index.

B. Item Deletion

```
Data Delete (Data D, PHR-tree)
LeafNode = Choose_leaf (Data D, PHR-tree);
DeletePointer (Data D, LeafNode);
Current Node = Leaf Node;
While Count(Current Node) < r do
        SibNode = argmin node belongs to Sib(Current
        Node)Count(Node);
        if (Count(Current Node) + Count(Sib Node) < R
        then
                Merge Node(Current Node, Sib Node);
                Current Node = PatentNode(Current
                Node);
        end if
end while
```

The item deletion to be conducted in a PHR-tree node includes both deletion operations on its R-tree node and Perfect Hash Index. The item deletion operation in the Perfect Hash Index is very easy procedure. Unlike the standard Bloom filter that cannot support the deletion operation because a bit 1 is likely to be set by multiple items, a counting Bloom filter is the one that effectively supports inserting, deleting, and querying items by replacing a bit in a standard Bloom filter with a counter. But in our case Perfect Hashing support the deletion and so no need to use extra counter for the store the count. But in the perfect hashing there is no need to update counter whenever an item is inserted or deleted. We first find the leaf node that contains the item to be deleted by using Choose Hash function. PHR-tree structure item is deleted from the node and also from the Perfect Hash Index. Here, we use dynamic Perfect Hashing that frees the Hash Index Space so that there is no space wastage or no empty item in the Perfect Hash Index. PHR-tree further deletes the pointer to item a in the leaf node. Due to the item deletion, the number of items at the current leaf node may be smaller than a predefined minimum bound r. So, PHR-tree will do the node merging operation, which combines two nodes that have fewer entries into a new one.

V. PERFORMANCE OF MULTIPLE QUERIES

A. Point Query

Point query determines whether a queried item is in a given PHR-tree structure or not. The query result can guide us to obtain actual data-related information from pointer address in a leaf node. We can carry out point query with O(1) complexity only in the root, which can generate an immediate result with a relatively higher probability of false positives inherently originated from Bloom filters. In contrast, performing a query with O(logN) complexity in the critical path from the root to a leaf node can ensure membership presence of an item. Using the computation of hash functions,

TABLE I. COMPARISON OF DIFFERENT INDEX STRUCTURE BASE ON VARIOUS COMPLEXITIES

Index Structure	Time Complexity	Space Complexity	Search Time
R-tree	O(nlog n)	O(n)	O(log n)
SD-tree	O(nm)	O(n)	O(logpf+a)
BR-tree	O(nlog n)	O(n)	O(log n)
PHR-tree	O(nlog n)	O(n)	O(log n)

we can check the counters of the corresponding Perfect Hash Index which shows the point query algorithm for an item with multidimensional attributes. If we keep the instruction in the dashed box, the algorithm complexity is O(1) by only checking the Perfect Hash Index of the root for item a. Since the root in a PHR-tree structure takes the union operation of its descendants in Perfect Hash Index, the union operations usually produce extra false positives. To get an exact query result, we can remove the dashed box instruction and the algorithm complexity becomes O(logN) since we need to check nodes in a path from the root to a leaf node in the worst case.

B. Range Query

The main function of this algorithm is to provide item identities whose attributes fall into the request bounds of a range query. We start the algorithm from the root of PHR-tree. Given a PHR-tree, we carry out a two-step process to implement the range query. In the first step, we search sub trees that intersect the queried range Q with p-dimensional attributes. If a CurNode has intersection with Q, it implies that its children may intersect Q as well. Thus, its child nodes will be recursively checked in the branch. Otherwise, we continue the check operation on its sibling nodes. The second step is linked to the leaf nodes whose MBRs intersect request Q.

C. Performance Analysis of Multiple Queries

PHR-tree support range query as well as point query. Other index structures that support range query and point query SD-tree, BR-tree and Distributed segment tree (DST). In SD-tree range query search time is O(log pf +a), while BR-tree and DST both takes O(log n). PHR - tree outperform all other index structure with O(log n) time. While searching tree for the range query our structure direct search for the value and apply hash function for the search of range time is O(1) and height of tree is O(logn) time to search the node is O(log n). Our query result come in O(log n) time. For the point query R-tree does not support point query. SD-tree and DST takes O(log n)time. BR-tree takes O(log n) time because it has bloom filters that reduce the cost of the searching. PHR tree uses perfect hashing and directly use hash function for the search of point data. PHR tree performs better than all other structure that supports the point query. While insertion takes same time in all the index structures and space complexity is also almost same in all O(n).PHR-tree structure performs well in both range query and point query as compared to the other index structure. Here aim is to support point query on the R-tree mean structure for multidimensional data support range query as well as point query efficiently.

VI. CONCLUSION AND FUTURE SCOPE

Here we discussed PHR-tree (Perfect Hash Base R tree) which supports multiple queries on the multidimensional data. It efficiently supports Point query on the multidimensional data, and range query. Use of perfect hashing instead of bloom filter gives more efficient result for the point query and also supports range query efficiently. Unlike bloom filter, there is no possibility of collision and return of false positive. PHR-tree keeps consistency between the data and attribute bound in the structure. In future this structure can expand for the different type of data like images, videos etc.

REFERENCES

[1] R. Devine, Design and Implementation of DDH: A Distributed Dynamic Hashing Algorithm, Proceedings of the 4th International Conference Foundation of Data Organizations and Algorithms, 1993,101-114.

[2] D. Ficara, S. Giordano and S. Kumar, B. Lynch,Divide and Discriminate:Algorithm for fast and deterministic hash lookups,Proceedings of the ACM/IEEE symposium on Architecture for networking and communication systems, ACM, 2002, New York, USA.

[3] C. du Mouza, W. Litwin, P. Rigaux, SD-Rtree: A Scalable Distributed Rtree, Proceedings International Conference of Data Engineering, 296-305, 2007.

[4] V. Gaede and O. Gu¨ nther, Multidimensional Access Methods, ACM computing surveys, 1998, 30, 2, 101-114.

[5] E. Bertino, B.C. Ooi, R. Sacks-Davis and K.-L. Tan and J. Zobel and B. Shidlovsky, and B. Cantania, Indexing Techniques for Advanced Database Applications, Kluwer Academics, 1997.

[6] H.V. Jagadish, B.C. Ooi, Q.H. Vu and R. Zhang and and A. Zhou,VBI-Tree: A Peer-to-Peer Framework for Supporting Multi-Dimensional Indexing Schemes, Proceedings International Conference on Data Engineering, 2006.

[7] H.V. Jagadish, B.C. Ooi, Q.H. Vu and R. Zhang, and A. Zhou,BATON: A Balanced Tree Structure for Peer-to-Peer Networks, Proceedings in International Conference Very Large Data Bases, 2005, 661-672.

[8] V. Markl ,MISTRAL: Processing Relational Queries using a Multidimensional Access Technique,Infix Verlag, 1999 .

[9] L. Arge, M. de Berg, H.J. Haverkort, K. Yi,The Priority R-Tree: A Practically Efficient and Worst-Case Optimal R-Tree, Proceedings ACM SIGMOD, 2004, 347-358.

[10] Timos K. Sellis, Nick Roussopoulos, Christos Faloutsos,the R+-Tree: A Dynamic Index for Multi-Dimensional Objects, Proceedings 13th International Conference on Very Large Data Bases, 1987, 507-518.

[11] N. Beckmann, H. P. Kriegel, R. Schneider, B. Seeger,The R*-tree: an efficient and robust access method for points and rectangles, Proceedings SIGMOD International Conference on Management of Data, 1990, 322-331.

[12] A. Guttman,R-trees: A Dynamic Index Structure for Spatial Searching, Proceedings ACM SIGMOD, 1984, 47-57.

[13] C. Zheng, G. Shen, S. Li, and S. Shenker,Distributed Segment Tree: Support of Range Query and Cover Query Over DHT, Proceedings International Workshop Peer-to-Peer Systems(IPTPS), 2006.