BR-Tree: A Scalable Prototype for Supporting Multiple Queries of Multidimensional Data

Yu Hua, *Member*, *IEEE*, Bin Xiao, *Member*, *IEEE*, and Jianping Wang, *Member*, *IEEE*

Abstract—Multidimensional data indexing has received much research attention recently in a centralized system. However, it remains a nascent area of research in providing an integrated structure for multiple queries on multidimensional data in a distributed environment. In this paper, we propose a new data structure, called BR-tree (Bloom-filter-based R-tree), and implement such a prototype in the context of a distributed system. The node in a BR-tree, viewed as an expansion from the traditional R-tree node structure, incorporates space-efficient Bloom filters to facilitate fast membership queries. The proposed BR-tree can simultaneously support not only existing point and range queries, but also cover and bound queries that can potentially benefit various data indexing services. Compared with previous data structures, BR-tree achieves space efficiency and provides quick response ($\leq O(log\ n)$) on these four types of queries. Our extensive experiments in a distributed environment further validate the practicality and efficiency of the proposed BR-tree structure.

Index Terms—BR-tree, multidimensional data, point query, range query, cover query, bound query.

1 Introduction

Distributed computing, especially resource-constrained systems, potentially requires space-efficient storage structures to promptly respond to data operations and efficiently support complex queries on multidimensional data items, such as *point query* which is to determine whether a given item is a member of a data set or not, *range query* which finds all items whose attribute values exist in the range of a query request, *cover query* which finds all ranges that can cover a randomly given point, and *bound query* which finds approximate but tight bounds of multidimensional attributes of an existing item in a data set.

The performance of such queries heavily depends on the provision of fast, highly scalable, and space-efficient query services. To support fast query service and improve system scalability, hash-based distributed structures, e.g., Distributed Hash Table, are studied in [1], [2], [3]. Such traditional single dimensional data structures can only support exact-matching point query since range attribute information is stripped when hash computations are executed. Although Group-Hierarchical Bloom filter Array (G-HBA) [4] and RBF [5] were implemented for distributed and fast point query, they failed to provide multiple-query services, while inaccurate query results may be returned due to false positives in Bloom filters.

In current emerging network applications such as environmental monitoring and geographical information systems [6], [7], queries invariably seek information about items having multiple attributes. In these cases, traditional single dimensional data structures are highly space inefficient as the accuracy of their query results depends upon maintaining the actual values of item identities and their attributes.

Multidimensional data indexing structures in a centralized system [8], [9], [10] have received considerable attention over years to facilitate data storage, management, and manipulation. Although R-tree structure [7] can support range query on multidimensional data items very well, it cannot support point query efficiently since R-tree only maintains the bounding boxes of multidimensional attributes and the pointers to actual data. Even though an item identity and its multiple attributes are provided, leaf nodes in the R-tree must store item identities to get point query result. This, in turn, requires a large storage space when the amount of data items is large. Though Bloom filter [11] structure is a space-efficient design for point query, it cannot support range query, cover query, or bound query services since it uses hash-based computation and has no multidimensional range information of stored items.

This paper presents a Bloom-filter-based R-tree (BR-tree), which integrates Bloom filters into R-tree nodes. The BR-tree is essentially an R-tree structure to support dynamic indexing, in which each node maintains a range index to indicate the attribute range of existing items. BR-tree takes advantage of the fact that range and cover queries are related in that an item viewed as the answer to a range query can trigger a partial solution for a cover query. This means 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 and bound query call similar operations to, respectively, obtain the existence of queried data and approximate bounds of attributes. Thus, in the BR-tree, the range query and cover query are supported following the branch of R-tree while point query and bound query are mostly served in the branch of Bloom filters.

- *Y. Hua is with the School of Computer, Huazhong University of Science and Technology, Wuhan, China, 430074. E-mail: csyhua@mail.hust.edu.cn.*
- *B. Xiao is with the Department of Computing, Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong, China. E-mail: csbxiao@comp.polyu.edu.hk.*
- *J. Wang is with the Department of Computer Science, City University of Hong Kong, Kowloon, Hong Kong, China. E-mail: jianwang@cityu.edu.hk.*

Manuscript received 2 Jan. 2009; accepted 24 Apr. 2009; published online 16 July 2009.
Recommended for acceptance by B. Veeravalli.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2009-01-0002.
Digital Object Identifier no. 10.1109/TC.2009.97.

In this paper, we focus on the design and implementation of the BR-tree data structure. Our contributions are summarized as follows:

- First, we propose a new data structure, BR-tree, for representing multidimensional data items and supporting the aforementioned four types of queries on such items. A BR-tree node contains multidimensional attribute ranges to facilitate range and cover queries, and an extra Bloom filter to improve query efficiency and accuracy for point and bound queries. Practical algorithms are given to carry out item insertion, point query, range query, cover query, and bound query operations in the BR-tree. To the best of our knowledge, this is the first paper to exploit the advantages of space-efficient Bloom filter and multidimensional R-tree to provide multiple lookup services in an integrated structure.
- Second, this paper presents a distributed and scalable BR-tree system to handle multiple queries with short query latency and high query accuracy where each BR-tree can handle the queries locally. We also provide a simple and effective strategy to allow nodes to update their stale information in the context of the distributed environment.
- Third, we show the query efficiency by providing applied query operations with low computational complexity. In a BR-tree with n nodes, all queries can be answered in $O(\log n)$ steps. We implement a BR-tree prototype system and test the system with real data traces, like HP trace, BU-Web trace, Forest CoverType trace, and Generating SpatioTemporal Data sets (GSTD) trace in a cluster of 30 network nodes. The experimental results showcase the query accuracy and storage efficiency.

The rest of this paper is organized as follows: We review and compare the related work in Section 2. In Section 3, we present the proposed BR-tree structure. Section 4 illustrates practical algorithms on the BR-tree and the complexity of query operations. Section 5 shows the method of deploying BR-tree in a distributed environment. Section 6 presents the BR-tree prototype implementation and displays experimental results. Finally, Section 7 concludes our paper.

2 Related Work

In this section, we briefly describe previous work in three areas relevant to the proposed new data structure BR-tree: Bloom filter, R-tree, and related tree structures supporting distinct queries.

A Bloom filter is a space-efficient data structure to store an index of an item and can represent a set of items as a bit array using several independent hash functions [11]. Using a Bloom filter to represent a set, one can make a point query in $O(1)$. The Bloom filter allows false positives in membership queries [12], though they occur in a very small probability.

Bloom filters provide a useful tool to assist network route lookup [13], packet classification [14], and the longest prefix matching [15]. All of these applications, however, have used Bloom filters mainly for items with a single attribute. There have been data structures that have made use of Parallel Bloom filters (PBF) [16] to provide network services for items with multiple attributes. However, PBF cannot efficiently support range and cover queries [17]. Other forms of Bloom filters that have been proposed for various purposes include counting Bloom filters [18], compressed Bloom filters [19], Group-Hierarchical Bloom filter Array [4], space-code Bloom filters [20], spectral Bloom filters [21], multidimension dynamic Bloom filters [22], and incremental Bloom filters [23].

The R-tree structure [7] 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 [24] and R^*-tree [25], 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 [26].

A lot of work which aims to support range query efficiently has been done [27], [28], [29], [30], [31], [32], [33]. 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 BR-tree, e.g., supporting two distinct queries in a unified structure [5], [8], [29], [34], [35]. BR-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 BR-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 [34] 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 [35], 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) [29] 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 [8] intends to support point and window (range) queries over large spatial data sets 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.

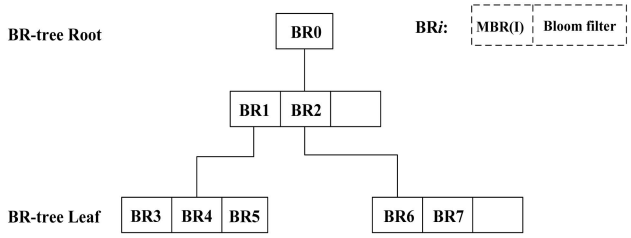

Fig. 1. An example of Bloom-filter-based R-tree structure.

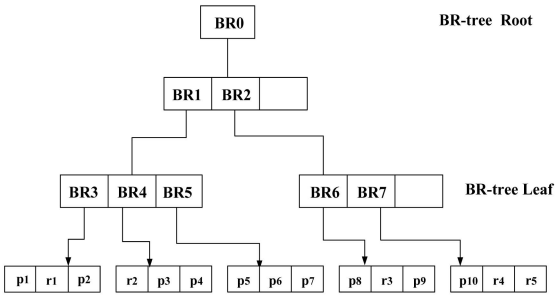

Fig. 2. A BR-tree example.

3 BR-Tree Structure

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

3.1 Proposed Architecture

A BR-tree is composed of *root, internal, and leaf* nodes. Fig. 1 shows an example of the proposed BR-tree structure. A BR-tree node combines an R-tree node with an extra Bloom filter where a Bloom filter is an m-bit array representing a set with n items by applying k-independent hash functions $\{h_1, \ldots, h_k\}$ on the item set. Because an R-tree node can exhibit a series of multidimensional attribute ranges and a Bloom filter 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 Bloom filter.

The *root* node (e.g., BR0) represents domain ranges of all possible attributes. Let R be the maximum number of children of a node. Each *internal* node can contain r ($\frac{R}{2} \leq r \leq R$) child nodes. We set a lower bound on r to prevent tree degeneration and to ensure an efficient storage utilization. Whenever the number of children drops below r, the node will be deleted and its children will be redistributed among sibling nodes. The upper bound R can guarantee that each tree node, in fact, can be stored exactly on one disk page. Each internal node contains entries in the form of $(I, Bloomfilter,\ Pointer)$ where $I = (I_0, I_1, \ldots, I_{p-1})$ is a p-dimensional bounding box, representing an MBR as shown in Fig. 1. I_i is a bounded interval, which can cover items in the ith dimensional space. $Pointer$ is the address of a child node. $Bloomfilter$ stores all hashed values of item identities, whose multidimension attributes are covered by the bounding box I.

An internal node (e.g., BR1) can illustrate the boundaries of a p-dimensional bounding box and the pointer to the addresses of its child nodes, and represent item identities covered by the bounding box.

All *leaf* nodes (e.g., BR3, etc.) appear at the bottom level and differ from internal nodes with the form $(I, Bloomfilter, item\text{-}pointer)$ where $item\text{-}pointer$ stores item identities and their pointer addresses. BR-tree allows the stored item to be either a point item or range for multiple queries. From the union of child nodes, we get the bounding range of the parent node in each dimension. The range union of siblings from the same level spans the whole range as the root does. This guarantees the data integrity in the BR-tree.

3.2 Load Balance

Similar to the traditional R-tree, the BR-tree is a load-balanced tree. Conventional approaches to storing items with multiple attributes divide a range into multiple segments and insert items into a segment covering their attributes. As a result, some segments may represent too many items and become overloaded. Nonetheless, conventional approaches are not suitable for a Bloom-filter-based structure because overloaded Bloom filters are susceptible to producing high false positive probabilities. BR-tree aims to be a load-balanced tree such that the nodes in the same level have approximately the same number of items. The load-balance property can efficiently decrease the false positive probability of Bloom filters in BR-tree nodes. A BR-tree based on an R-tree reconfigures the segments of a multidimensional range after using bounding boxes to cover items. This guarantees that the BR-tree nodes in the same level contain approximately the same number of items.

3.3 Example for Multiple Queries

Fig. 2 exhibits an example of BR-tree structure. Our current data set, represented as a BR-tree with root node $BR0$, contains two subsets, $BR1$ and $BR2$, respectively, having subsets, $BR3, BR4, BR5$ and $BR6, BR7$. We store data objects (represented as points p) and ranges (represented as ranges r) into our BR-tree structure. Fig. 3 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 BR-tree become very simple and can be fast implemented compared with

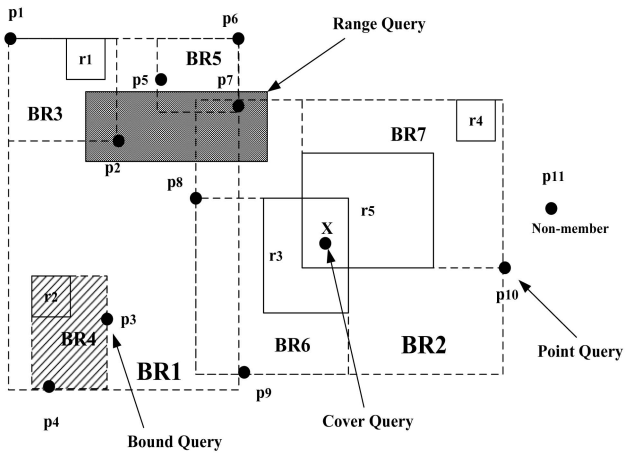

Fig. 3. An example of multiple queries in a BR-tree.

```
Choose_Leaf (Item a, BR-tree)
CurNode = RootNode(BR-tree);
if (CurNode == LeafNode) then
   Return CurNode;
else
   Subtree= ChooseLeastMBREnlargement(CurNode, Item
   a);
   Choose_Leaf (Item a, Subtree);
end if
```

Fig. 4. The algorithm to locate a leaf node for item a.

previous R-tree structures. For example, if we need to know whether item $p10$ belongs to our data set, we need to check the Bloom filters along the query path from node $BR0$, $BR2$, to $BR7$ by computing the hash values of item $p10$. Bloom filters will return positive to the existence of $p10$ in this example. Given the outside $p11$ in Fig. 3, the Bloom filters will return negative of its existence after the computation of hash functions for item $p11$. 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, or $O(\log n)$ to eliminate the false positive by multistep verifications on Bloom filters along the query path in the BR-tree.

The processing of a range query starts from the root. If there is a node entry whose *MBR* intersects the query region, its subtree is recursively explored. When the range query encounters a leaf node, we get all items whose bounding rectangles intersect the query region. For example, the shaded region in Fig. 3 intersects *MBRs* of both leaf nodes, $BR3$ and $BR5$. As a result, items $p2$ and $p7$ will be returned for the range query.

A cover query is to obtain all multidimensional ranges covering a given item. For example, given an item X in Fig. 3, a cover query can determine that the two-dimensional bounding ranges $r3$ and $r5$ can cover it after query operations along the path from $BR0$, $BR2$ to $BR6$ and $BR7$ 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 Bloom filters 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. 3, we know that $p3$ is contained in the leaf node $BR4$. Thus, the shaded area, i.e., $BR4$, 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 BR-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 BR-tree depth will become larger and more storage space will be required.

4 Local Operations on a BR-Tree

This section introduces practical operations applied on a BR-tree in response to an item insertion and deletion, point query, range query, cover query, and bound query. A BR-tree needs to be updated when new items arrive and thus can correctly respond to multiple-query requests from users. Note that our proposed algorithms here only show the local operations that indicate how to obtain query results from a BR-tree in a network node.

```
Insert_Item (Item a, BR-tree)
LeafNode = Choose_Leaf (Item a, BR-tree);
if Entry(LeafNode) ≥ R then
   LeafNode=Quadratic_split(LeafNode);
end if
Insert(Item a, LeafNode);
CurNode = LeafNode;
while CurNode ≠ NULL do
   if a ∉ MBR(CurNode) then
      ExpandMBR(CurNode)
   end if
   Insert(Item a, BloomFilter(CurNode));
   CurNode = ParentNode(CurNode);
end while
```

Fig. 5. The algorithm of inserting item a.

4.1 Item Insertion

Insertion of an item into a BR-tree includes operations on the R-tree and corresponding Bloom filters. Since an inserted item needs to be placed in a leaf node, we need to first locate the leaf node and then insert it. Fig. 4 shows the algorithm to locate a leaf node for a new arrival item a. We use $CurNode$ to denote a currently checked BR-tree node. The suitable leaf node for the item can be found in $O(\log n)$ time, by examining a single path as shown in the R-tree design [7].

Fig. 5 presents the insertion algorithm when adding an item a into our BR-tree structure. 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 Bloom filters in the leaf node and all its ancestors till the root. This process is in $O(\log n)$ time complexity. Otherwise, we need to split the leaf node by utilizing the quadratic-cost algorithm [7], [25], 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 Bloom filter.

4.2 Item Deletion

The item deletion to be conducted in a BR-tree node includes both deletion operations on its R-tree node and Bloom filter. The item deletion operation using Bloom-filter-based structure is deemed as a difficult problem, though some possible solutions exist [12]. 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 [18] is the one that effectively supports inserting, deleting, and querying items by replacing a bit in a standard Bloom filter with a counter. When an item a is inserted or deleted, its associated counters are increased or decreased by one accordingly.

Fig. 6 shows the deletion operation on a BR-tree for an item a. We first find the leaf node that contains the item to be deleted by using $Choose\_Leaf$ function. The node

```
Delete_Item (Item a, BR-tree)
LeafNode = Choose_Leaf (Item a, BR-tree);
DecreaseHashedCounter(1, BloomFilter(LeafNode));
DeletePointer(Item a, LeafNode);
CurNode = LeafNode;
while Entry(CurNode) < r do
   SiblingNode = arg min_{node∈Sibling(CurNode)} Entry(node);
   if Entry(CurNode + SiblingNode) < R then
      CombineNode(CurNode, SiblingNode);
      CurNode = ParentNode(CurNode);
   end if
end while
```

Fig. 6. The algorithm of deleting item a.

```
CombineNode (nodeN(A), nodeN(B))
for (i = 1; i ≤ p; i++) do
   N(A)_i = MaxInterval(N(A)_i, N(B)_i);
end for
InsertItemPointers(N(B), N(A));
BF(N(A)) = UnionBloomFilter(N(A), N(B));
Return N(A);
```

Fig. 7. The algorithm of merging two BR-tree nodes.

```
Boolean Member (Item a, BR-tree node)
Flag = True;
for (j = 1; j ≤ k; j++) do
   if Hash_j(a) == 0 in the Bloom filter of node then
      Flag = False, Break;
   end if
end for
Return Flag;
```

Fig. 8. The function to check the presence of item a.

deletion on the Bloom filter in the located leaf node is done by decreasing every counter by 1 by applying the counting Bloom filter [18]. BR-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 threshold r. Consequently, BR-tree will proceed with the node merging operation, which combines two nodes that have fewer entries into a new one. Fig. 7 illustrates node merging algorithm to produce a node with the maximized MBR and a unioned Bloom filter.

4.3 Point Query

Point query allows us to determine whether a queried item a is a member in a given BR-tree structure. 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. To know the presence of an item in a BR-tree node, we have the *Boolean* function *Member* as shown in Fig. 8. Using the computation of hash functions, we can check the counters of the corresponding counting Bloom filters.

Fig. 9 shows the point query algorithm for an item with multidimensional attributes. If we keep the instruction in the

```
Boolean Point_Query (Item a, BR-tree)
CurNode = Root, Flag = False;
while (Member(a, CurNode)) do
   - - - - - - - - - - - - - - - - - - - -
   || Flag = True, Return Flag;      ||
   - - - - - - - - - - - - - - - - - - - -
   if CurNode == LeafNode then
      Flag = True, Break;
   end if
   CurNode = ChildNode(CurNode);
end while
if Flag then
   Flag = Verify_Item_by_identity(a);
end if
Return Flag;
```

Fig. 9. The algorithm of point query.

```
Range_Query (Range Request Q_[1,...,p], Root)
CurNode = Root;
while CurNode! = LeafNode do
   if Intersect(CurNode, Q) then
      CurNode = ChildNode(CurNode);
   else
      CurNode = SiblingNode(CurNode);
   end if
   Range_Query(Q, CurNode);
end while
InsertLeafItems(Items, Result, Q);
Return Result
```

Fig. 10. The algorithm of range query.

dashed box, the algorithm complexity is $O(1)$ by only checking the Bloom filter of the root for item a. Since the root in a BR-tree structure takes the union operation of its descendants in Bloom filters, 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.

4.4 Range Query

Fig. 10 shows the range query algorithm in the BR-tree. The main function of this algorithm is to provide item identities whose attributes fall into the request bounds of a range query. All qualified items will be included in an item set $Result$ that is initialized to be $\emptyset$. We start the algorithm from the root of BR-tree. Given a BR-tree, we carry out a two-step process to implement the range query. In the first step, we search subtrees 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.

4.5 Cover Query

Fig. 11 shows the cover query algorithm for an item with multidimensional attributes, which will return a set of multidimensional range objects. The returned range objects can cover the given item identity a. Different from the range query algorithm, we only check nodes covering a. In the

Cover_Query (Item a, Root)

$CurNode = Root$;
while $CurNode! = LeafNode$ **do**
  if $Contain(CurNode, a)$ **then**
    $CurNode$ = ChildNode($CurNode$);
  else
    $CurNode$ = SiblingNode($CurNode$);
  end if
  $Cover\_Query(a, CurNode)$;
end while
InsertLeafRangeObjects($RangeObjects$, $Result$, Item a);
Return $Result$

Fig. 11. The algorithm of cover query.

MBR Bound_Query (Item a, BR-tree)

$CurNode = Root$;
while $CurNode! = LeafNode$ **do**
  if $Member(a, CurNode)$ **then**
    $CurNode$ = ChildNode($CurNode$);
  end if
end while
if $Member(a, LeafNode)$ **then**
  Return MBR($LeafNode$);
end if

Fig. 12. The algorithm of bound query.

BR-tree, a node that can cover a multiattribute item a means its MBR containing a. If MBR of an internal node (e.g., $CurNode$) cannot enclose a, neither can be done for its descendant nodes because an MBR is formed from the union operation on descendant nodes. The checking process will be recursively carried out to some leaf nodes where some range objects can be found. All range objects that can cover item a are inserted into the set $Result$.

4.6 Bound Query

Bound query can exhibit approximate range information of multidimensional attributes of a queried item. This assists fast attribute bound estimation but avoids frequent and direct data access. The bound query can obtain its result by executing similar operations as the point query does. The main difference is that bound query returns the multidimensional ranges indicated by the MBR of a leaf node. Point query, however, determines whether the queried item is a member of a data set.

Fig. 12 shows the bound query algorithm. Note that in the bound query, the given item a must exist in one leaf node in a BR-tree. Thus, we should locate the leaf node that makes $Member(a, LeafNode)$ to be true. To attain it, we need to traverse all child nodes whose $Member(a, CurNode)$ results are true in the BR-tree. After finding the right leaf node, we return its multidimensional ranges represented by MBR as the tight attribute bounds to item a.

4.7 Simple Summary of Multiple Queries

The four types of queries discussed above exhibit distinct query requests from users in real-world applications. Point query generally receives the most attentions due to its wide application scope. Range query has been widely applied in spatial database. Cover query and bound query, although until recently they have been rarely touched, play a pivotal role in emerging distributed applications, such as PetaByte-scale distributed storage system and large-scale environment monitoring. There are two major differences between cover query and bound query. First, the queried item can be arbitrary in the cover query while it must be an existing one in the queried data set in the bound query. Second, the returned ranges must be explicitly stored in the data set in the cover query while the bound query does not require such knowledge.

The proposed BR-tree is a combination of Bloom filters [11] with R-tree by integrating Bloom filters into R-tree nodes, which is not trivial because BR-tree maintains the advantages of both Bloom filters and R-tree and further facilitates mixed queries, like point query followed by bound query and range query if the point query replies positive.

In a BR-tree, point and bound queries execute similar operations in the Bloom filters stored in BR-tree nodes. The difference is that after completing the item presence check in a leaf node, point query returns the data while bound query returns the MBR range information indicated by the associated leaf node. Range and cover queries also carry out similar operations. After completing the range check in a leaf node, range query needs to return all stored data that are represented as points in the leaf node while cover query returns the stored range objects.

4.8 Comparisons of BR-Tree and Other State-of-the-Art Structures

BR-tree is different from other state-of-the-art structures, including Bloom filter [11], baseline R-tree [7], BATON [35], VBI-tree [34], DST [29], SD-Rtree [8], and RBF [5]. BR-tree can achieve comprehensive advantages. BR-tree has a bounded $O(\log n)$ complexity for point query. The Bloom filter in the root of BR-tree can provide fast query result with $O(1)$ complexity. However, the result may not be accurate due to false positive. In applications requiring exact query results, we can follow the Bloom filter branch of BR-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. Since Bloom filters 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 bit checking on Bloom filters can directly probe the same counters, saving much query time. Because we mainly follow the R-tree part in BR-tree to obtain range and cover query services, these queries have the same complexity as R-tree to be $O(\log n)$. Meanwhile, BR-tree structure can support bound query by checking the Bloom filters along query path from the root to a leaf node, achieving $O(\log n)$ complexity. Bloom filter structure is a space-efficient design, which is also adopted in the BR-tree. The BR-tree structure is also able to support distributed queries as described in the next section in detail.

No existing architectures provide the aforementioned four types of queries for multidimensional data. BATON and VBI-tree aim to provide *virtual* indexing frameworks. Their practical performance heavily depends on the underlying used structures, not themselves. Although RBF is able to support point query, its query result is probabilistic (not exact matching). Normally, the baseline R-tree cannot

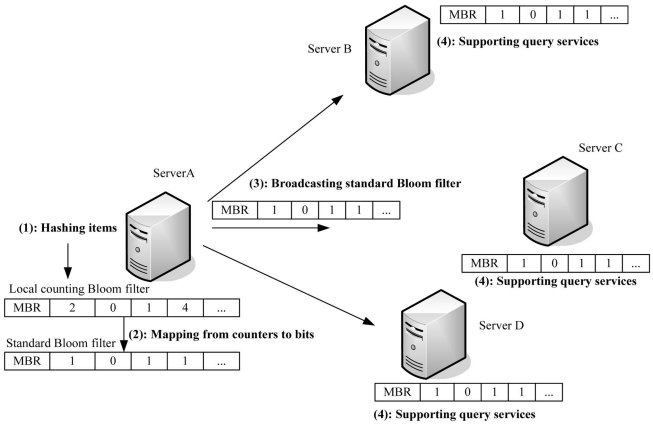

Fig. 13. A scenario depicting the $BR\text{-}tree$ replica broadcasting from server A in a distributed system.

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 BR-tree with baseline R-tree in terms of point query, with SD-Rtree in terms of point and range queries, and DST in terms of range and cover queries. The experimental results have been shown in Section 6 by testing real data traces.

5 Distributed BR-Trees

A large-scale distributed system consisting of many servers potentially requires a distributed structure to facilitate multiple-query scheme, while providing system scalability in an efficient way. Hence, we exploit the simple and space-efficient characteristics of BR-tree and deploy it in multiple servers. The distributed BR-tree shows its advantages for easy deployment, scalability, and feasibility. In this section, we will first present distributed BR-tree structure. Then, we will show the query operations in the distributed context and describe how to update stale replicas to assure accurate services.

5.1 Distributed BR-Tree Structure

The basic idea of our distributed BR-tree scheme is to locally maintain replicas of all BR-tree roots distributed at servers, which constitute a *BR-tree Array*. A replica, called *BR-tree vector*, of a BR-tree root has the same MBR as the root, but a standard Bloom filter transformed from a counter-based Bloom filter. Thus, a nonzero value in a counter position is transformed into bit 1 in the same bit position of the standard Bloom filter. Otherwise, the bit 0 will be set.

The distributed BR-tree structure allows each server (treated as a *network node*) to store a BR-tree representing its local items and the replicas of BR-tree roots representing items in all other (remote) servers. In a large-scale distributed system composed of n nodes, a network node needs to store n BR-tree replicas in the BR-tree array, i.e., $(n-1)$ replicas from other $(n-1)$ nodes and one from itself. Fig. 13 shows an example to exhibit BR-tree replica information broadcasting among four servers. Each data structure in a server is composed of two parts, the root of a local BR-tree and BR-tree array (i.e., the replica table, serving as an instantaneous local mirror reflecting the most recent data information in other nodes). Thus, a local query on a single network node can obtain the global query result through executing operations on a local BR-tree and stored replicas. Since each network node only needs to maintain replicas of BR-tree roots of other nodes, the deployment does not require much storage space, and hence, we can place BR-tree array entirely into memory to obtain fast query services.

5.2 Operations on Distributed BR-Trees

Distributed BR-tree deployed in each server can efficiently support query services, including point, range, cover, and bound queries, through explicitly indicating "local hit" on the local BR-tree root and BR-tree replica array (except replica of the local BR-tree). A "local hit" refers to the membership existence on Bloom filters for point and bound queries or the intersection or inclusion of MBRs for range and cover queries. If a local hit happens on the local BR-tree root, a query result can be obtained from the local BR-tree. Local hits taking place in the BR-tree array can trigger request forwarding to remote nodes indicated by their hit replicas, requiring further search on those nodes. Thus, queries on the local BR-tree can directly obtain the lookup results and those on replicas of other nodes can help to select remote servers quickly and precisely. The reply from the remote server can be sent back as the query results.

The distributed BR-tree structure needs to carry out, besides query operations, stale content update operation to assure high query accuracy. The update operation order is issued from a local BR-tree to its replicas distributed in other nodes. Because a local BR-tree uses the counter-based Bloom filter and its replicas use standard Bloom filter, the value change in a counter may not trigger corresponding bit change. For instance, a counter increment or decrement (nonzero) maps to the same 1 in the standard Bloom filter. Thus, we need to count the number of real changes (from 1 to 0 or 0 to 1) made to its replica from a local BR-tree. Only when the number of changes is over a predefined threshold since the last update, the stale update message will be sent out. This threshold value reflects data staleness degree and links to query accuracy. Note that replicas in all other $(n-1)$ nodes contain exactly the same information as the local one at the time of the last update, assuming that the last update operation has been performed successfully and correctly in each node.

5.3 Update Stale Replicas

In a distributed system where multiple BR-tree servers are deployed for the purpose of scalability, the process of updating data information among them becomes critical in providing accurate and reliable query results. Since the content of a local BR-tree root may change dynamically and the updating process takes time due to the network latency, we need to design a simple and efficient scheme to update stale replicas. The update must take place in both the Bloom filter and MBR parts in a replica.

The changes in the Bloom filter part can trigger the replica update. We compute the percentage of stale bits in the local standard Bloom filter of a BR-tree. When the percentage is over a predefined threshold, a node needs to broadcast the update messages. The "stale bits" are those outdated bits in

the standard Bloom filter that are different from the dynamically changed counter-based Bloom filter. Stale bits appear when a BR-tree carries out item insertion or deletion. Inserting an item to a local BR-tree may increase a counter value from "0" to "1." The delayed updating on other replicas can result in *false negative* for membership query in a distributed environment. Stale bits in a replica may answer "no" to a membership query of an item, although it actually exists in a remote server. On the contrary, deleting an item may decrease a nonzero counter to zero and cause *false positive* in distributed lookups. Stale bits in a local replica may answer "yes" to a membership query of an item, although it actually does not exist in a remote server. In practice, the penalty of false negative, which decreases query accuracy, imposes a larger impact on membership query than that of false positive. We still can eliminate false positive result by a double verification on a remote BR-tree. Thus, when performing the computation of the percentage of staleness, we assign stale bits linked to false negative with a larger weight than those with false positives.

The condition as when to send out a replica update can be defined similarly by computing the percentage of "stale space" in MBR because an MBR outlines multidimensional attribute ranges. The "stale space" refers to the widened or narrowed space presented by MBR due to inserting new items or deleting old ones. The widened space may lead to false negatives for range and cover queries, while the narrowed space may lead to false positives. Normally, the false negatives have a larger impact than false positive to get accurate results and a larger weight will be assigned to them. In addition, the larger the changed area, the larger the impact on the query accuracy. For example, a 10 percent space increment in MBR, if it is not updated timely in other nodes, often makes more requests to obtain false or incomplete answers than a 1 percent space increment when query requests follow a uniform distribution. In our following prototype implementation, we compute the percentages of stale bits and space, any of which is larger than a predefined threshold further triggering the replica update operations to guarantee the query accuracy.

6 Prototype Implementation

We have implemented the proposed BR-tree structure and tested operations, such as item insertion and multiple queries for multidimensional data. We further deployed BR-trees in a real cluster and used four traces to compare the performance of BR-tree with R-tree [7], SD-Rtree [8], and DST [29] in terms of query latency, accuracy, message overhead, and storage space.

6.1 Constructing Experiment Environment

We present the construction of our prototype implementation by first introducing used traces, then describing experiment settings in detail.

6.1.1 Available Traces

We utilize real-world and synthetic traces to comprehensively evaluate our proposed BR-tree structure. We first select real-world traces from two typical applications exhibiting the data locality access pattern, i.e., BU-Web-Client [36], [37] and HP file system [38], and then utilize the popular high-dimensional data set, *Forest CoverType* trace from the UCI machine learning repository website (http://archive.ics.uci.edu/ml/datasets.html). In order to further test our BR-tree for a data set following the uniform distribution, we utilize the GSTD [39], which is a data set generator, to artificially generate uniformly distributed data trace. Table 1 presents the overview of tested data traces in our performance evaluation where HP, BU-Web, and Forest CoverType traces exploit data locality and GSTD contains uniformly distributed data. Four tested traces are as follows:

TABLE 1
An Overview of Experimental Data

	Uniform distribution	Skew distribution
HP Trace	no	yes (locality)
BU-Web Trace	no	yes (locality)
Forest CoverType Trace	no	yes (locality)
GSTD Trace	yes	no

- *HP trace*: The HP file system trace [38] is a 10-day, 500 GB trace recording file system accesses from 236 users at the HP Lab. We selected three attributes, i.e., last "modified" time, "read" and "write" data amounts as the range attributes of queried files for range and cover queries. We concatenated the "device number" and "file name" to be file ID for point and bound queries.
- *BU-Web trace*: Internet traffic archive [36] in Lawrence Berkeley National Laboratory provides a web trace, BU-Web-Client [37]. We take the concatenation of a user request identity and its time stamp as the item identity to uniquely represent each item in our data source model.
- *Forest CoverType trace*: The *Forest CoverType* data set contains 581,012 data points and each has 54-dimensional attributes. Specifically, these 54-dimensional attributes include 10 quantitative variables, four binary wilderness areas and 40 binary soil type variables. Data records for forest cover types exhibit locality distribution. The detailed information of the data set can be found at the website [40].
- *GSTD trace*: We use standard GSTD generator [39] tool to artificially generate a three-dimensional $2{,}000 \times 2{,}000 \times 2{,}000$ data trace, following the uniform distribution. If the size of an item in the data set is smaller than 1/10 cell, the item is considered as a point object represented by its coordinates. Otherwise, it is viewed as a range object.

6.1.2 Implementation Details

We implemented the BR-tree structure on the Linux kernel 2.4.21 running on a cluster that has 30 nodes, each equipped with dual-AMD processors and 2 GB memory and connected with a high-speed network. The artificially generated querying points and ranges are distributed within a simulated data space, following either uniform or skew distribution. The skewed requests are generated using

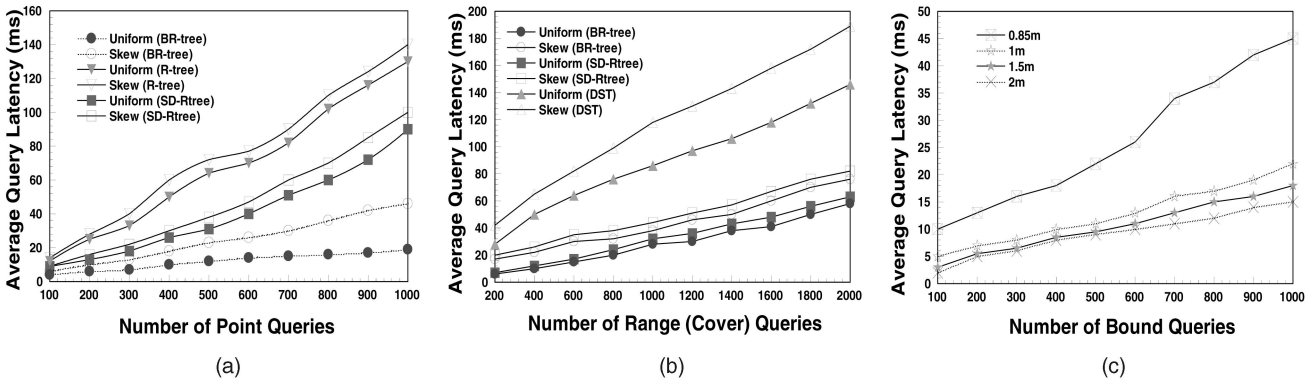

Fig. 14. Average query latency in the trace of HP file system. (a) Point query. (b) Range and cover queries. (c) Bound query.

a Zipfian distribution with parameter 1.0. Note that the bound query needs to check the MBR of an existing item, not a random item compared with other queries. Thus, we only use item IDs that have appeared in the testing to be the requests for bound query.

We designed the space for Bloom filters in BR-tree structures according to the basic requirement in [11]. Each Bloom filter has $m = \frac{kq}{\ln 2}$ bits when applying k hash functions for q items. Here, we use $k = 7$ hash functions. There are about 4 million files evenly distributed in 30 nodes using the HP trace. Hence, each standard Bloom filter tested in the HP trace requires $m = 160$ KB space to represent local files. Using the intensifying technique [4], [41], BU-Web, Forest CoverType, and GSTD contain around 10^6 items, thus requiring $m = 40$ KB space for a standard Bloom filter. In Bloom filters, we use MD5 as hash functions because of its well-known properties and relatively fast implementation. The MD5 hash functions allow an item identity to be hashed into 128 bits by calculating the MD5 signature. Afterwards, we divide the 128 bits into four 32-bit values and apply the modulo operation on the filter size.

To update stale replicas, we adopt offline computation on the percentage of stale bits and space as discussed in Section 5.3. The weight associated with false negatives is set to be 50 percent larger than that associated with false positives. We set the threshold of updating stale replicas to be 10 percent.

6.2 Performance Evaluation

To evaluate data structure effectiveness, we compare the proposed structure, BR-tree, with the baseline R-tree [7], SD-Rtree [8] for point query, SD-Rtree, and DST [29] for range and cover queries in terms of query latency, accuracy, message overhead, and storage space. Since only BR-tree can support bound query, we show the performance of distributed bound queries for BR-trees in 30 network nodes.

6.2.1 Query Latency

Figs. 14, 15, 16, and 17 show the average query latency using the HP file system trace, BU-Web trace, high-dimensional Forest CoverType data set, and artificial GSTD trace, respectively. The average query latency increases when there are more queries simultaneously submitted to distinct data structures in a distributed environment. Note that R-tree structure cannot efficiently support the point query for an item only from its own ID (except using brute-force searching approach), each queried item is given its multidimensional attributes. To locate an item at a leaf node in an R-tree, we follow the MBR branch that can match its multidimensional attributes to move to the leaf node.

Fig. 14a shows the latency of point query and we observe that BR-tree spends less time than baseline R-tree and SD-Rtree on completing point query no matter what distributions the query requests follow. There are two major reasons for this. First, although BR-tree, R-tree, and SD-Rtree have the same complexity of $O(\log n)$ for point query, BR-tree in practice only checks Bloom filters along

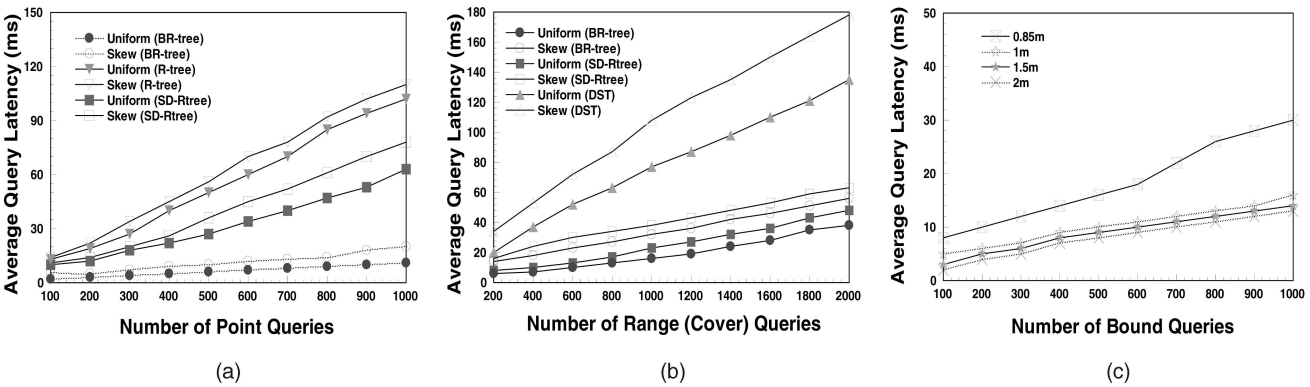

Fig. 15. Average query latency in the BU-Web trace. (a) Point query. (b) Range and cover queries. (c) Bound query.

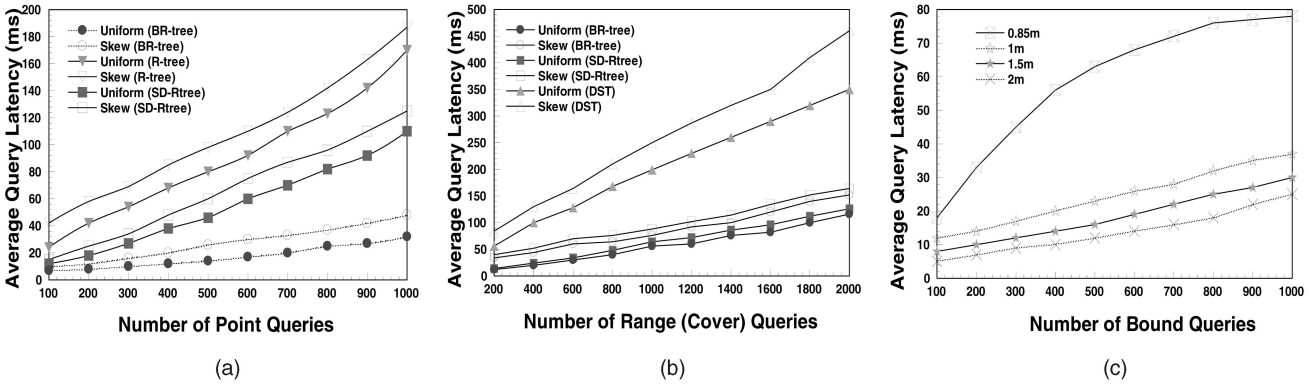

Fig. 16. Average query latency in the Forest CoverType trace. (a) Point query. (b) Range and cover queries. (c) Bound query.

the query path from the root to a leaf node by checking the same counter positions after computing MD5 hash functions only once. In contrast, R-tree and SD-Rtree have to check each MBR along the same query path and conduct matching-based operation in each node to determine the item's membership, thus requiring much more time. Second, R-tree and SD-Rtree can determine the item existence only after it checks a leaf node. In other words, R-tree and SD-Rtree have to complete the verification along the query path from the root no matter whether the queried item in fact exists or not. However, BR-tree can immediately get the negative answer from its root by checking on one Bloom filter, resulting in much shorter query latency. We also observe that the requests following the uniform distribution have much smaller latency than skewed requests. This is simply because the HP trace actually contains files exhibiting the access locality property, meaning that files have a nonuniform distribution in the data space. As a result, requests following the uniform distribution can contain more queried but nonexisted items. Consequently, BR-tree can give a quicker response based on the checking on the root Bloom filter.

Fig. 14b displays the query latency for range and cover queries. To get the query results, we need to compare MBRs along query paths (from the root to a leaf node) in different data structures. Note that here BR-tree and DST are evaluated by using 1,000 range and 1,000 cover queries while SD-Rtree being unable to support cover query is evaluated by using 2,000 range queries. It is observed that the query latency in SD-Rtree and BR-tree is close, but much smaller than that in DST. The main reason is that DST utilizes 2^N branch segment trees to extend the binary tree to maintain items that have N-dimensional attributes. Thus, to get a result for a given query, DST has to check multiple segment trees, which is a process resulting in a large latency.

Since bound query can be supported in the BR-tree, but not in the R-tree, we have the experimental result for bound queries implemented in distributed BR-trees as shown in Fig. 14c. Bloom filters were designed with storage space like $0.85m, 1m, 1.5m$, and $2m$ where $1m$ is the standard space requirement as discussed in Section 6.1.2. Fig. 14c illustrates that the query latency increases quickly when the allocated space for Bloom filters is set to be $0.85m$. Given such crowded space allocation, Bloom filters will report more false positives, leading to lengthy checking in both local and remote BR-trees. We also tested the latency by increasing the space size for Bloom filters to $1.5m$ and $2m$. The results show that the query latency can be slightly reduced due to false positive decrement. We further show the query latency by using the BU-Web trace that records real-world events in Fig. 15.

Fig. 16 shows the average query latency in a distributed environment using the high-dimensional data set, Forest CoverType trace. BR-tree can achieve the shortest query latency compared with R-tree, SD-Rtree, and DST in point, range, and cover queries. DST has the largest query latency because it needs to first locally execute range split operation (a time consuming process), and then carry out DHT-based *get* operation.

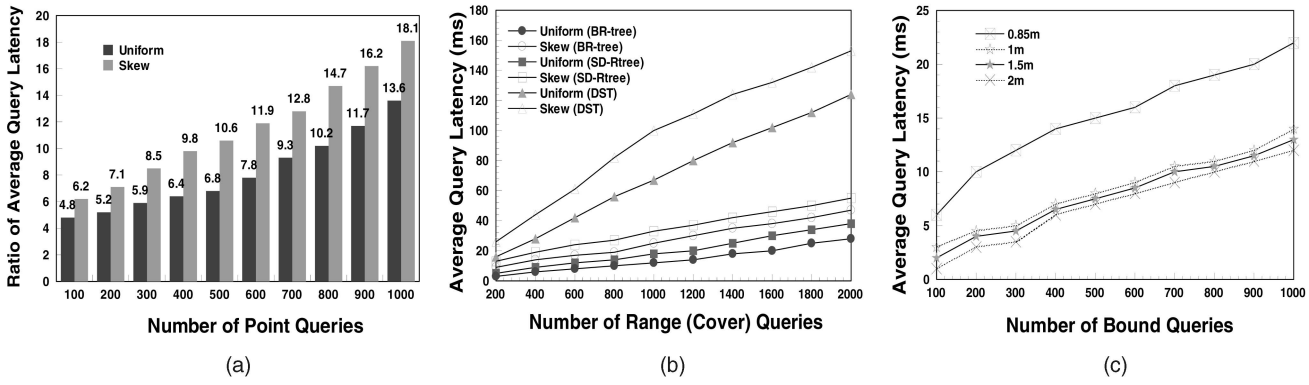

Fig. 17. Average query latency in the GSTD trace. (a) Point query. (b) Range and cover queries. (c) Bound query.

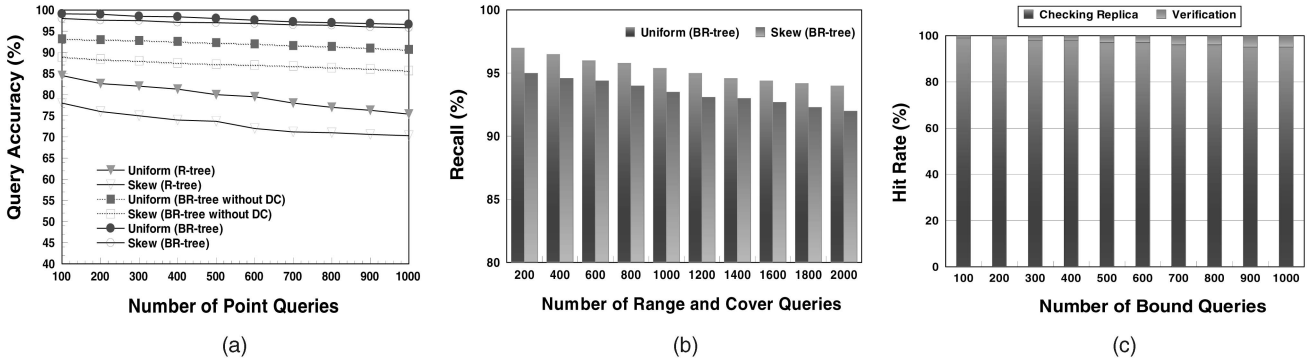

Fig. 18. Query accuracy in BR-tree structure when using the HP trace. (a) Point query. (b) Range and cover queries. (c) Bound query.

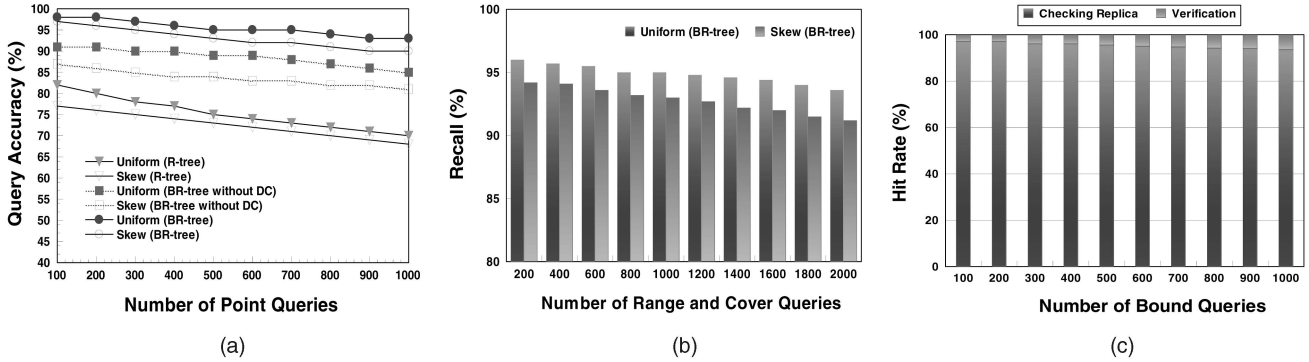

Fig. 19. Query accuracy in BR-tree structure when using the Forest CoverType trace. (a) Point query. (b) Range and cover queries. (c) Bound query.

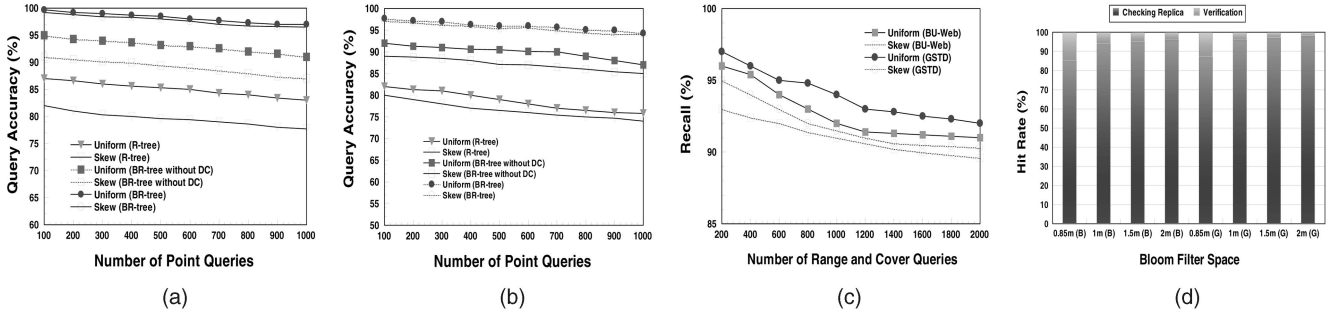

Fig. 20. Query accuracy in BR-tree structure when using the BU-Web and GSTD traces. (a) Point query using BU-Web. (b) Point query using GSTD. (c) Range and cover queries. (d) Hit rate for 1,000 bound queries.

Fig. 17 shows the latency comparisons by using an artificial GSTD trace for 1,000 query requests. Fig. 17a plots the ratio of baseline R-tree to BR-tree with regard to average point query latency. BR-tree can achieve time savings by a factor of 18.1 and 13.6 compared with R-tree, respectively, for requests following skew and uniform distribution. Figs. 17b and 17c show the average latency results for range, cover, and bound queries that are consistent to other tested traces.

6.2.2 Query Accuracy

In a distributed environment, queries may not get correct results due to stale information scattered among network nodes or data structures themselves. We can improve the query accuracy of BR-trees by applying the *Double Checking* (DC) mechanism from which we need to verify the presence of a queried item in the leaf node that contains the real item ID. We conducted experiments to show the accurate query results provided by BR-trees for multiple queries and made comparisons with R-tree structure in Figs. 18, 19, and 20 using four tested data traces.

Fig. 18 shows the query accuracy using the HP trace. Fig. 18a illustrates the point query accuracy for R-tree structure, BR-tree with and without the double checking function. All files accessed in the HP trace are evenly stored among all 30 nodes and each file has a single local copy. The "query accuracy" (i.e., hit rate) denotes the percentage of true hits, in which distributed BR-trees reply only one BR-tree storing the queried item (i.e., one hit), among all point queries for actually existing items. Zero or multiple hits mean a false hit. We observe that only using Bloom filters in BR-trees without DC the hit rate can be around 90.1 percent. Using double checking on leaf nodes can improve this rate up to 98.2 percent in 1,000 query requests. The improved accuracy cannot rise to 100 percent in practice. We conjecture that the main reason is that every network node in the distributed experimental environment maintains the stale information of item memberships.

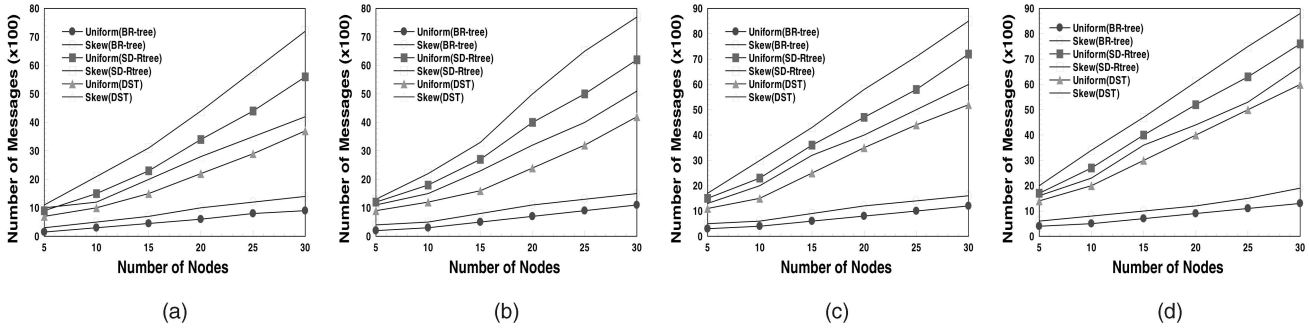

Fig. 21. Number of exchanged messages in all tested data traces. (a) HP trace. (b) BU-Web trace. (c) Forest CoverType trace. (d) GSTD trace.

We evaluate the accuracy of range and cover queries by examining their "recall" rate in Fig. 18b. Given a query q, let $T(q)$ be the set containing all items for query q, and $A(q)$ be the set of items returned by using the BR-tree scheme. $recall$ is defined as $recall = \frac{|T(q) \cap A(q)|}{T(q)}$. We can use the brute-force approach in parallel in all nodes to obtain the $T(q)$. The "recall" rate of range and cover queries is over 90 percent, but cannot reach 100 percent because staleness of replicas limits the query accuracy.

Fig. 18c illustrates the hit rate for bound queries to obtain the MBRs of existing items. Distributed BR-trees can provide exact-matching (100 percent) accuracy for existing items by simply displaying MBRs from either a local BR-tree or a remote one. However, some items may be deleted from the system and due to update delay, item deletion may not be timely broadcasted to other remote nodes. A local BR-tree response to bound queries can get 97.2 percent correct results by checking its Bloom filter in the root node, as shown in Fig. 18c. A remote BR-tree verification can complement the bound query, securing a correct reply. We further examined the query accuracy of BR-trees by using high-dimensional data set, i.e., Forest CoverType trace, as shown in Fig. 19.

Fig. 20 shows the query accuracy using the BU-Web and GSTD traces. The experimental results validate our proposed BR-tree structure and its scalability. Compared with standard R-trees, distributed BR-trees can provide more accurate replies to distinct query requests, achieving over 90 percent accuracy in most cases.

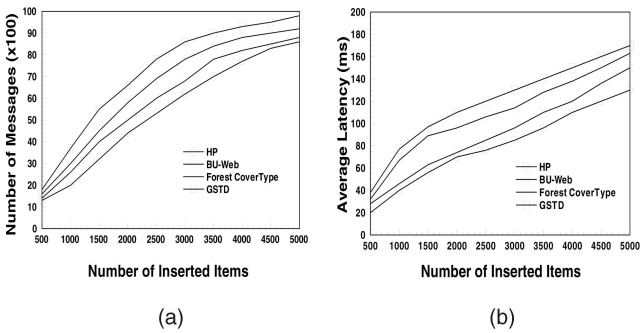

Fig. 22. Insertion operation. (a) Number of messages. (b) Average latency.

6.2.3 Message Overhead

We evaluate the communication overhead and scalability of BR-trees by measuring the number of messages exchanged to respond to 1,000 multiquery requests when the number of nodes increased from 5 to 30. Fig. 21 shows the comparisons between BR-tree, SD-Rtree, and DST by counting the number of total messages (e.g., updating stale replicas, locating a network node) in tested data traces. The number of messages in BR-tree is not very sensitive to distinct query requests. This is because each node contains approximately equal number of items, hence achieving the load balance. SD-Rtree requires more messages due to its verification forwarding scheme to avoid the address error [8]. DST, on the other hand, utilizes each segment tree for representing each dimension and its update operations are more sensitive to the changes of stale information, thus generating more message overhead.

Specifically, we test the communication overhead and average latency when we insert items into distributed BR-trees, totally 30 of them, as shown in Fig. 22. Item insertion requires to first locate an appropriate network node whose MBR covers the item, and then insert it into a leaf node. More items inserted, longer the time to finish the insertion process. The insertion of an item may cause the local node containing updated information in its Bloom filter and MBR. Thus, further messages are needed to update stale replicas in other remote nodes.

6.2.4 Storage Space

We compare the required storage space of BR-tree with SD-Rtree and DST. Table 2 shows the comparative results that are normalized to the space used by the BR-tree structure. An SD-Rtree needs to maintain a global image of all distributed trees locally. A DST adds 2^N branch segment trees to support N-dimensional query services, thus requiring a large storage space. Compared with SD-Rtree and DST, BR-tree is a space-efficient data structure that can support multiquery services for multidimensional data.

TABLE 2
Used Space Normalized to BR-Tree

	BR-tree	SD-Rtree	DST
HP Trace	1.0	2.6	8.1
BU-Web Trace	1.0	2.8	9.3
Forest CoverType Trace	1.0	3.7	11.6
GSTD Trace	1.0	3.2	9.8

7 Conclusion

In this paper, we proposed a Bloom-filter-based R-tree structure, i.e., BR-tree, for supporting multiple queries of items having multidimensional attributes. The proposed BR-tree can efficiently support point, range, cover, and bound queries. To the best of our knowledge, we are the first to address the bound query. Note that bound query could be widely applied into real applications that do not have the exact-matching requirement. The BR-tree structure makes it possible for fast point query and accurate bound query since BR-tree keeps the correlated consistency between queried data and their attribute bounds in an integrated structure. We also present how to deploy BR-trees in a distributed environment to provide scalable query services. The system prototype implementation demonstrates that BR-tree structure is scalable while providing accurate responses to distinct queries.

Acknowledgments

This work was supported in part by NSFC under Grant 60703046, HK RGC PolyU under Grant 5307/07E, CityU of HK under Grant 7002327 and 9681001, National Basic Research 973 Program of China (2004CB318201) and PCSIRT (IRT-0725). The authors greatly appreciate the GSTD tool, real-world traces provided by HP Labs, Lawrence Berkeley National Lab and UCI machine learning repository, and constructive comments from anonymous reviewers.

References

[1] R. Devine, "Design and Implementation of DDH: A Distributed Dynamic Hashing Algorithm," *Proc. Fourth Int'l Conf. Foundations of Data Organizations and Algorithms,* pp. 101-114, 1993.

[2] "Distributed Hash Tables Links," http://www.etse.urv.es/cpairot/dhts.html, 2009.

[3] M. Harren, J.M. Hellerstein, R. Huebsch, B.T. Loo, S. Shenker, and I. Stoica, "Complex Queries in DHT-Based Peer-to-Peer Networks," *Proc. Int'l Workshop Peer-to-Peer Systems (IPTPS),* 2002.

[4] Y. Hua, Y. Zhu, H. Jiang, D. Feng, and L. Tian, "Scalable and Adaptive Metadata Management in Ultra Large-Scale File Systems," *Proc. Int'l Conf. Distributed Computing Systems (ICDCS),* pp. 403-410, 2008.

[5] Y. Hua, D. Feng, H. Jiang, and L. Tian, "RBF: A New Storage Structure for Space-Efficient Queries for Multidimensional Metadata in OSS," File and Storage Technologies (FAST) Work-in-Progress Reports, 2007.

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

[7] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. ACM SIGMOD,* pp. 47-57, 1984.

[8] C. du Mouza, W. Litwin, and P. Rigaux, "SD-Rtree: A Scalable Distributed Rtree," *Proc. Int'l Conf. Data Eng. (ICDE),* pp. 296-305, 2007.

[9] V. Gaede and O. Günther, "Multidimensional Access Methods," *ACM Computing Surveys,* vol. 30, no. 2, pp. 170-231, 1998.

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

[11] B. Bloom, "Space/Time Trade Offs in Hash Coding with Allowable Errors," *Comm. ACM,* vol. 13, no. 7, pp. 422-426, 1970.

[12] A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," *Internet Math.,* vol. 1, pp. 485-509, 2005.

[13] A. Broder and M. Mitzenmacher, "Using Multiple Hash Functions to Improve IP Lookups," *Proc. IEEE INFOCOM,* pp. 1454-1463, 2001.

[14] F. Baboescu and G. Varghese, "Scalable Packet Classification," *IEEE/ACM Trans. Networking,* vol. 13, no. 1, pp. 2-14, Feb. 2005.

[15] S. Dharmapurikar, P. Krishnamurthy, and D.E. Taylor, "Longest Prefix Matching Using Bloom Filters," *Proc. ACM SIGCOMM,* pp. 201-212, 2003.

[16] Y. Hua and B. Xiao, "A Multi-Attribute Data Structure with Parallel Bloom Filters for Network Services," *Proc. IEEE Int'l Conf. High Performance Computing (HiPC),* pp. 277-288, 2006.

[17] B. Xiao and Y. Hua, "Using Parallel Bloom Filters for Multi-Attribute Representation on Network Services," *IEEE Trans. Parallel and Distributed Systems,* 2009.

[18] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary Cache: A Scalable Wide Area Web Cache Sharing Protocol," *IEEE/ACM Trans. Networking,* vol. 8, no. 3, pp. 281-293, June 2000.

[19] M. Mitzenmacher, "Compressed Bloom Filters," *IEEE/ACM Trans. Networking,* vol. 10, no. 5, pp. 604-612, Oct. 2002.

[20] A. Kumar, J.J. Xu, J. Wang, O. Spatschek, and L.E. Li, "Space-Code Bloom Filter for Efficient Per-Flow Traffic Measurement," *Proc. IEEE INFOCOM,* pp. 1762-1773, 2004.

[21] C. Saar and M. Yossi, "Spectral Bloom Filters," *Proc. ACM SIGMOD,* pp. 241-252, 2003.

[22] D. Guo, J. Wu, H. Chen, and X. Luo, "Theory and Network Application of Dynamic Bloom Filters," *Proc. IEEE INFOCOM,* 2006.

[23] F. Hao, M. Kodialam, and T.V. Lakshman, "Incremental Bloom Filters," *Proc. IEEE INFOCOM,* pp. 1741-1749, 2008.

[24] T.K. Sellis, N. Roussopoulos, and C. Faloutsos, "The R^+-Tree: A Dynamic Index for Multi-Dimensional Objects," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* pp. 507-518, 1987.

[25] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles," *Proc. ACM SIGMOD,* pp. 322-331, 1990.

[26] C. Bohm, S. Berchtold, and D.A. Keim, "Searching in High-Dimensional Spaces Index Structures for Improving the Performance of Multimedia Databases," *ACM Computing Surveys,* vol. 33, no. 3, pp. 322-373, 2001.

[27] J. Aspnes and G. Shah, "Skip Graphs," *Proc. ACM-SIAM Symp. Discrete Algorithms (SODA),* pp. 384-393, 2003.

[28] A.R. Bharambe, M. Agrawal, and S. Seshan, "Mercury: Supporting Scalable Multi-Attribute Range Queries," *Proc. ACM SIGCOMM,* pp. 353-366, 2004.

[29] C. Zheng, G. Shen, S. Li, and S. Shenker, "Distributed Segment Tree: Support of Range Query and Cover Query Over DHT," *Proc. Int'l Workshop Peer-to-Peer Systems (IPTPS),* 2006.

[30] "Opendht," http://opendht.org/, 2009.

[31] J. Gao and P. Steenkiste, "An Adaptive Protocol for Efficient Support of Range Queries in DHT-Based Systems," *Proc. IEEE Int'l Conf. Network Protocols (ICNP),* pp. 239-250, 2004.

[32] D. Li, J. Cao, X. Lu, K.C.C. Chan, B. Wang, J. Su, H. va Leong, and A.T.S. Chan, "Delay-Bounded Range Queries in DHT-based Peer-to-Peer Systems," *Proc. Int'l Conf. Distributed Computing Systems (ICDCS),* 2006.

[33] X. Li, Y.J. Kim, R. Govindan, and W. Hong, "Multi-Dimensional Range Queries in Sensor Networks," *Proc. ACM Conf. Embedded Networked Sensor Systems (SenSys),* pp. 63-75, 2003.

[34] H.V. Jagadish, B.C. Ooi, Q.H. Vu, R. Zhang, and A. Zhou, "VBI-Tree: A Peer-to-Peer Framework for Supporting Multi-Dimensional Indexing Schemes," *Proc. Int'l Conf. Data Eng. (ICDE),* 2006.

[35] H. Jagadish, B. Ooi, and Q. Vu, "BATON: A Balanced Tree Structure for Peer-to-Peer Networks," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* pp. 661-672, 2005.

[36] "The Internet Traffic Archive," http://ita.ee.lbl.gov/, 2009.

[37] C.A. Cunha, A. Bestavros, and M.E. Crovella, "Characteristics of WWW Client-Based Traces," Technical Report TR-95-010, Dept. of Computer Science, Boston Univ., 1995.

[38] E. Riedel, M. Kallahalla, and R. Swaminathan, "A Framework for Evaluating Storage System Security," *Proc. Conf. File and Storage Technologies (FAST),* pp. 15-30, 2002.

[39] Y. Theodoridis, J. Silva, and M. Nascimento, "On the Generation of Spatiotemporal Datasets," *Proc. Int'l Symp. Spatial Databases (SSD),* pp. 147-164, 1999.

[40] The Forest CoverType Data Set, "UCI Machine Learning Repository,"http://archive.ics.uci.edu/ml/datasets/Covertype, 2009.

[41] Y. Zhu, H. Jiang, J. Wang, and F. Xian, "HBA: Distributed Metadata Management for Large Cluster-Based Storage Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 19, no. 6, pp. 750-763, June 2008.

Yu Hua received the bachelor's and PhD degrees in computer science from Wuhan University, China, in 2001 and 2005, respectively. He was a research assistant in the Department of Computing at Hong Kong Polytechnic University in 2006. Now he is an assistant professor in the School of Computer at Huazhong University of Science and Technology, China. His research interests include distributed computing and network storage. He is a member of the IEEE and the IEEE Computer Society.

Bin Xiao received the BSc and MSc degrees in electronics engineering from Fudan University, China, in 1997 and 2000, respectively, and the PhD degree from the University of Texas at Dallas in 2003 in computer science. Now he is an assistant professor in the Department of Computing of Hong Kong Polytechnic University, Hong Kong. His research interests include distributed computing systems, data storage, and security in wired and wireless computer networks. He is a member of the IEEE.

Jianping Wang received the BSc and the MSc degrees in computer science from Nankai University, Tianjin, China, in 1996 and 1999, respectively, and the PhD degree in computer science from the University of Texas at Dallas in 2003. She is currently an assistant professor at the Department of Computer Science, City University of Hong Kong. Her research interests include optical networks and wireless networks. She is a member of the IEEE.