LETTER

Efficient Predicate Matching over Continuous Data Streams

Hyeon-Gyu KIM[†a)], ***Member***, **Woo-Lam KANG**[†], **Yoon-Joon LEE**[†], ***and*** **Myoung-Ho KIM**[†], ***Nonmembers***

SUMMARY In this paper, we propose a predicate indexing method which handles equality and inequality tests separately. Our method uses a hash table for the equality test and a balanced binary search tree for the inequality test. Such a separate structure reduces a height of the search tree and the number of comparisons per tree node, as well as the cost for tree rebalancing. We compared our method with the IBS-tree which is one of the popular indexing methods suitable for data stream processing. Our experimental results show that the proposed method provides better insertion and search performances than the IBS-tree.

key words: *data streams, predicate index, IBS-tree*

1. Introduction

There are many applications to monitor continuous streams of data items (relational tuples) such as auction bids, stock exchanges, network measurements, web page visits, sensor readings and so on [1]. These applications commonly involve a large number of queries which consist of *interval predicates* [3]. For example, in a stock trading application, many queries with interval predicates reflecting user interests are dynamically registered and used to monitor the change of stock prices.

One of the common approaches to efficient processing of a large number of these queries is to use a *predicate index* [2]. Many predicate indexing methods have been introduced so far. Among them, only main memory-based indexes such as the *interval binary search tree* (IBS-tree) [2], the *CEI-based index* [3] and the *VCI index* [4] can be used for stream applications, because most of them are designed to run on main memory to provide their responses in a timely manner.

The CEI-based index assumes the range of input values is known in advance and decomposes the range into fixed-size segments (Fig. 1 (a)). Each segment is again decomposed to non-overlapping smaller-size segments. A predicate interval is mapped to one or more consecutive segments. However, its fixed structure may degrade insertion performance significantly, especially when handling open intervals that can be distributed to a large number of segments. The VCI index has the similar mechanism and can be suffered from the fixed nature of its index structure.

On the other hand, the IBS-tree determines the segments dynamically based on given predicate intervals (Fig. 1 (b)). So, its insertion performance can be better than above two methods. The insertion (or deletion) performance is important for real-time responses of stream applications where predicates can be registered or unregistered dynamically. Below, we only consider the IBS-tree.

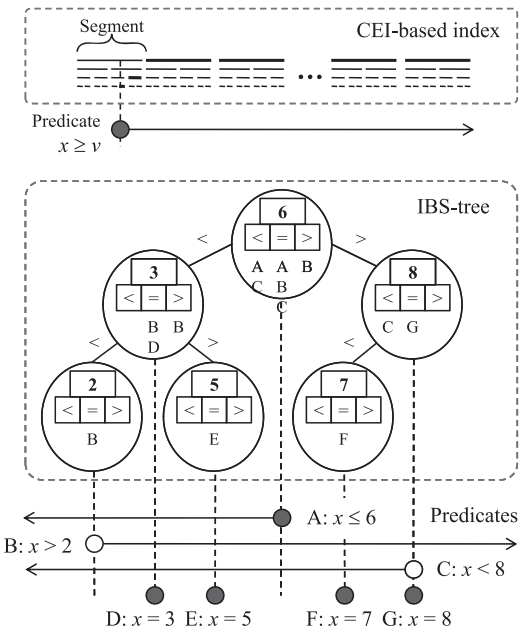

Fig. 1 (a) CEI-based index and (b) the IBS-tree.

Figure 1 (b) shows an example of the IBS-tree, given 7 predicates from *A* to *G*. For each distinct endpoint of a predicate interval, there is a tree node that consists of three slots, each of which has a set of predicates whose ranges are smaller than, equal to and greater than the endpoint, respectively. However, it doesn't need to keep the "=" slot in every node, because a probability that an input tuple satisfies the equality test becomes very low if the value range is large enough. We also observed that maintaining the "=" slot in every node can significantly degrade insertion performance.

In this paper, we propose a predicate indexing method which improves performance by dealing with the equality and inequality tests separately.

2. Our Method

The proposed method uses a hash table for the equality test and a balanced binary search tree for the inequality test. Figure 2 shows an example of our method, given same predi-

Manuscript received May 15, 2009.

[†]The authors are with the Division of Computer Science, the School of Electrical Engineering and Computer Science, KAIST, 373–1 Kusong-dong, Yusong-gu, Taejon, 305–701, South Korea.

a) E-mail: hgkim@dbserver.kaist.ac.kr

DOI: 10.1587/transinf.E92.D.1787

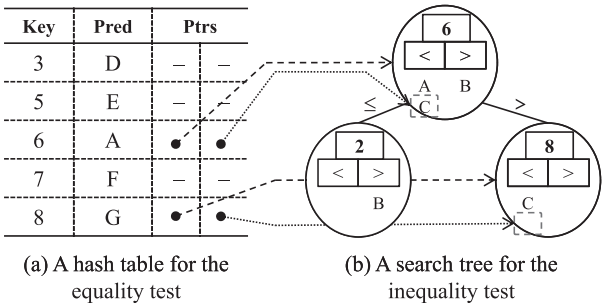

Fig. 2 Structure of the proposed index.

cates of Fig. 1 (b). The hash table has an entry for each distinct value of *equality* predicate endpoints. Each entry has a set of predicates with the same endpoint (i.e., column *Pred*) and pointers to a tree node and the predicates of its "<" slot (i.e., column *Ptrs*). (Their usages will be discussed below.)

On the other hand, our search tree has a node for each distinct value of *inequality* predicate endpoints. Its node consists of two slots, each of which has a set of predicates whose ranges are smaller than and greater than the endpoint, respectively. Since the tree is constructed from inequality predicates only, its height becomes smaller than the IBS-tree. In addition, the average number of comparisons per node reduces because the equality test is not conducted in each node. These characteristics enable our method to show better search performance than the IBS-tree.

Whenever an input tuple arrives, we first find the hash table, then the search tree. In the tree search, if its value is equal to a node, it is routed to its left subtree. In this process, predicates in the "<" slot of the node are added to a result. However, this may lead to inaccurate search results. For example, suppose that a tuple (with value) 8 arrives in Fig. 2. Then, the search will end in the "<" slot of node 8, and the predicates *B*, *C* and *G* will be returned as a result. But, *C* must be excluded from the result because it is defined as "$x < 8$".

To avoid such inaccuracy, we decompose predicates in the "<" slot into two parts: predicates whose endpoints are equal to a node value and those whose endpoints are not. For convenience, we call the former *primary* (smaller-than) predicates and the latter *secondary* predicates. In Fig. 2 (b), *A* is a primary predicate of node 6, while *C* is a secondary one. If an input tuple v arrives, node v is organized to return secondary predicates, not whole ones in its "<" slot.

To enable this, we keep an additional hash entry for each endpoint v of smaller-than predicates. We set pointers of the entry (i.e., those in column *Ptrs* of Fig. 2 (a)) to its corresponding tree node v and its secondary predicates. Whenever a tuple v arrives, we check if its hash entry has a pointer to node v. If so, we organize the node v to return its secondary predicates (by switching pointers). Then, we conduct the tree search. After the search ends, we restore the node v to return its original predicates. Note that in our method, the search always ends at a leaf node, while it may stop at an intermediate node in the IBS-tree.

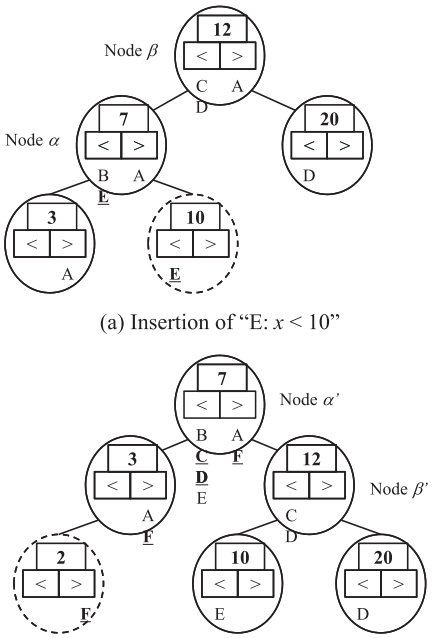

Fig. 3 Insertion of predicates to our search tree.

Now, let us discuss the insertion of predicates in our method. To illustrate the process, consider a search tree constructed from 4 predicates "$A : x > 3$", "$B : x < 7$", "$C : x < 12$" and "$D : x < 20$". We would insert a new smaller-than predicate "$E : x < 10$" to the search tree (Fig. 3 (a)). To do this, we first add a new node with value 10 to the search tree, and assign *E* to the "<" slot of the node as well as the slot of node 7. We also assign a new entry for *E* to the hash table and connect pointers to node 10 and its predicates.

Note that, if we assign a predicate to the "<" slot of a node, we don't have to assign it to the "<" slots of left descendants of the node. This is clear from that "$x < v$" logically implies "$x < u$" if node v is a left descendant of node u. Consequently, there is no overlap of predicate assignments in the "<" slots of nodes u and v. On the other hands, the ">" slot of a node includes all predicates of the ">" slots of its left descendants. The interval "$x > v$" cannot be covered by the interval "$x > u$". In this case, if we assign a predicate to node v, we also need to assign it to node u together.

Let $left(v)$ and $right(v)$ be the sets of predicates in the "<" and ">" slots of node v, respectively. Then, the above characteristics of our search tree can be described as follows.

Theorem 1. *Predicate assignments in our search tree satisfy the following two properties.*

(1) If v is a left descendant of u, then
$left(u) \cap left(v) = \emptyset$ and $right(u) \supseteq right(v)$
(2) If v is a right descendant of u, then
$right(u) \cap right(v) = \emptyset$ and $left(u) \supseteq left(v)$

Proof. The proof for property (1) is done by above description. Property (2) is symmetric to property (1), so we omit its proof. □

Now, let us consider the insertion of "$F : x > 2$" in the status of Fig. 3 (a). The insertion of F incurs unbalance of the tree: the left subtree of node 12 outweighs its right subtree. To rebalance the tree, any of the existing algorithms such as the *AVL tree* and the *Red-black tree* can be used; we assume that one of them is used and our tree is rebalanced based on it.

Note that the rebalancing incurs the redistribution of predicates in our search tree. In Fig. 3, predicates of nodes α and β need to be redistributed. (Actually, it is the case of *LL rotation* in the AVL-tree scheme.) A rule for the redistribution can be easily derived from Theorem 1. In this case, node β becomes a right child of node α after rebalancing the tree. Therefore, we can use property (2) for the redistribution as follows.

$$right(\beta') = right(\beta) - right(\alpha)$$
$$left(\alpha') = left(\alpha) \cup left(\beta)$$

We can have the redistribution rule for the symmetric case from Theorem 1. Here, we do not describe complete rules from the lack of available spaces.

The IBS-tree has two more rules [2] in addition to the above ones, which are required for redistribution of predicates in the "=" slots. The rules can be described as follows. Below, $middle(v)$ denotes a set of predicates in the "=" slot of node v. (There are also rules for the symmetric case, which we omit in this paper.)

$$middle(\beta') = middle(\beta) - right(\alpha)$$
$$middle(\alpha') = middle(\alpha) \cup left(\beta)$$

As a result, whenever a tree is rebalanced, the IBS-tree needs to update four slots, while our method updates two slots. We observed through our experiments that the update cost is high. These things make our method outperform the IBS-tree regarding the insertion performance.

Other features of insertion algorithm are similar to that of the IBS-tree. To insert a predicate with a closed interval "$a \le x \le b$", we decompose it into two parts "$a \le x$" and "$x \le b$". For each part, we again separate the equality condition "$a = x$" (or "$b = x$") and add a hash entry for the condition. Then, we add a new node for "$a < x$" (or "$x < b$") to the search tree. To handle a smaller-than predicate such as "$a < x$", we use an algorithm *AddLeft* which is shown in Fig. 4. (There is also an algorithm *AddRight* for the greater-than part "$x \le b$".) The algorithm is the same as that of the IBS-tree, except that it does not include the equality test in Step (4) and (9). In the algorithm, the function *rightUp* returns the lowest ancestor of node n in the tree that contains n in its left subtree. *leftEnd* and *rightEnd* denote the left and right endpoints a and b, respectively. After Step (12), tree rebalancing and predicate redistribution are conducted as discussed above.

Since our method uses both of the hash table and the search tree, the storage complexity of our method becomes larger than that of the IBS-tree. Let the total number of predicates be N. Then, the storage complexity of the IBS-tree is $O(NlogN)$ in the worst case because a predicate can place in up to $logN$ tree nodes. In the same manner, our search tree requires $O(NlogN)$ storage in the worst case. Our hash table requires $O(N)$ storage because it can keep entries for all predicates. Thus, the storage complexity of our method becomes $O(N + NlogN)$ in the worst case.

```
AddLeft(node n, predicate p)
(1)  IF n = NULL
(2)      Create a new node with value p.leftEnd FI
(3)  IF n.value = p.leftEnd
(4)      IF rightUp(n) < p.rightEnd
(5)          Add p to the ">" slot of node n FI
(6)  ELSE IF n.value < p.leftEnd
(7)      AddLeft(n.rightChild, p)
(8)  ELSE
(9)      IF rightUp(n) < p.rightEnd
(10)         Add p to the ">" slot of node n FI
(11)     AddLeft(n.leftChild, p)
(12) FI
```

Fig. 4 Insertion of the left-end of a predicate to our search tree.

However, the actual size of our index structure can be smaller from the different constant factors. In our method, a hash entry uses 12 bytes of memory for storing the pointer to a predicate list (4 bytes) and two pointers to a tree node and its secondary predicates (8 bytes). A tree node uses larger memory (28 bytes) for a predicate ID (4 bytes), balance information (8 bytes), pointers to smaller-than and greater-than predicate lists (8 bytes) and pointers to the left and right subtrees (8 bytes). Therefore, the total size of our index structure can be calculated as $12N + 28NlogN$ bytes. On the other hand, the IBS-tree consumes $32NlogN$ bytes of memory, since each node of the tree has one more pointer (to the equality predicate list) than our search tree. If N is given to 1,000,000 and all of the values are distinct, the IBS-tree requires about 640 Mbytes of memory, while our method requires about 572 Mbytes.

3. Experimental Results

In this section, we provide experimental results that compare the IBS-tree and our method in terms of insertion and search performances. To conduct experiments, we implemented algorithms of both methods as well as a data generator to synthesize test data sets. We set the domain of predicate endpoint values to [0, 1000000] and randomly produced up to 100,000 predicates with their inequality types (i.e., "<", "≤", ">" and "≥"). We excluded equality predicates to make search trees of both methods have the same number of nodes.

As a hash table, we currently use an array whose size is large enough to cover the domain. Assuming the above domain, the array uses 4 Mbytes of memory since each element has a pointer to a hash entry (4 bytes). If there are 100,000 predicates, our method uses 6.8 Mbytes smaller than the

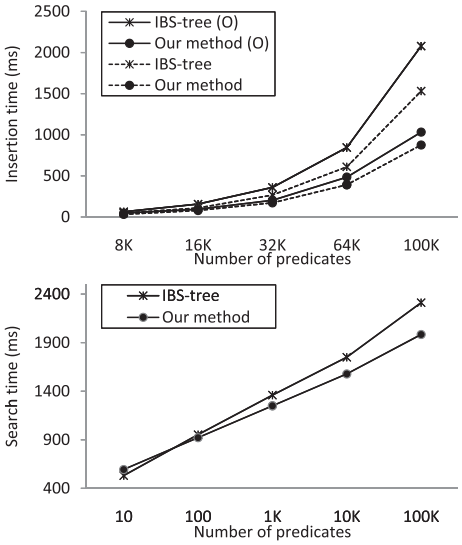

Fig. 5 Our method vs. IBS-tree: (a) insertion performances (upper) and (b) search performances (lower).

IBS-tree as discussed in Sect. 3. Thus, it is affordable to use 4 Mbytes for the array. Note that, if a conventional hashing method is used, the search performance of our method can be degraded by the cost of overflow handling. Our experiments were conducted on Intel Pentium IV 3.2 GHz machine, running Window XP, with 2 G main memory.

First, we would compare insertion performances of two methods. We used two kinds of data sets: one has only open intervals with types "≤" and "≥", and the other has both of open and closed intervals with types "<", "≤", ">" and "≥". In the latter, we organized the ratio of two kinds of intervals to be the same. We increased the number of predicates from 8 K to 100 K and observed their insertion times of both cases. In Fig. 5 (a), experimental results for the former data set are marked as solid lines, while the results for the latter one are marked as dashed lines. In both of the results, insertion performance of our method is approximately 2 times better than that of the IBS-tree. This is because our method does not have the "=" slot in each tree node, while the IBS tree does. The result also infers that the cost of updating a slot is high. Note that insertion times of experiments with the former data set are larger than those of experiments with the latter one. This is due to that the average number of predicates per node becomes larger when a tree is made from open intervals only, compared with the case when the tree is made from both open and closed intervals.

Then, we compared search performances of both methods. In this experiment, we increased the number of open interval predicates (with types "≤" and "≥") n from 10 to 100 K and conducted 10 million times of searches. The result (Fig. 5 (b)) shows that our method is about 15% better than the IBS-tree when n is larger than 1 K in our experimental setup. We also observed that our method is about 7% better in the extreme case when the domain size is equal to n, which we omitted from the lack of spaces. This shows that the benefit of the smaller number of comparisons per node of our search tree is bigger than that of the search stop at an intermediate node of the IBS-tree.

4. Conclusion and Future Work

In this paper, we proposed a predicate indexing method which uses a hash table for the equality test and a balanced binary search tree for the inequality test. By excluding slots for the equality test from the search tree, our method provides better performance than the IBS-tree. Our experimental results show that the proposed method provides approximately 100% and 15% better insertion and search performances, respectively.

As a hash table, we used an array that can cover the domain of input values. If a conventional hashing method is used, the search performance of our method can be degraded by the cost of overflow handling. To overcome this, we are planning to apply a caching mechanism to our method. We think it is acceptable because our method uses smaller memory than the IBS-tree.

Acknowledgements

This work was supported by Korea Science and Engineering Foundation (KOSEF) grant funded by the Korea government (MEST) (No.R0A-2007-000-10046-0).

References

[1] L. Golab and M.T. Ozsu, "Issues in data stream management," ACM SIGMOD, Record 32, no.2, pp.5–14, 2003.
[2] E. Hanson, M .Chaabouni, C. Kim, and Y. Wang, "A predicate matching algorithm for database rule systems," ACM SIGMOD, pp.271–280, 1990.
[3] K. Wu, S. Chen, and P.S. Yu, "Interval query indexing for efficient stream processing," ACM CIKM, pp.88–97, 2004.
[4] K. Wu, S. Chen, P.S. Yu, and M. Mei, "Efficient interval indexing for content-based subscription E-Commerce and E-Service," CEC-East, 2004.