A Hybrid Method for Fast Predicate Matching in Data Stream Processing*

HYEON GYU KIM AND MYOUNG HO KIM
Division of Computer Science, School of Electrical Engineering and Computer Science
Korea Advanced Institute of Science and Technology
373-1 Guseong-Dong, Yuseong-Gu, Daejeon 305-701, South Korea

Efficient predicate matching is essential to provide real-time responses in processing continuous data streams. Predicate indexes are commonly used to find the predicates matched by continuously arriving tuples in a timely manner. In this paper, we propose a hybrid indexing method to support fast insertion of predicates in data stream processing. Our method deals with open and closed interval predicates separately. We use shared ordered lists of the PB+-tree scheme to process open interval predicates efficiently, while use an algorithm of the IBS-tree to process closed interval ones. Our experimental results show that (1) the proposed method provides approximately 3 times better insertion performance than the IBS-tree when there are both of open and closed interval predicates with ratio of 1:1 and (2) their performance gap can be more than 12 times as the ratio of open interval predicates increases.

Keywords: Interval predicate indexing, Predicate matching, IBS-trees, Data streams

1. INTRODUCTION

Recent years have witnessed the emergence of a new class of applications, which monitor continuous streams of data items such as auction bids, stock exchanges, network measurements, web page visits, sensor readings, and so on [1,2,3]. These applications commonly involve a large number of queries which consist of interval predicates [4,5]. For example, in a stock trading applications, many queries with interval predicates reflecting user interests are dynamically registered and used to monitor the real-time change of stock prices.

One of the common approaches to processing the large number of queries efficiently is to use a predicate index. The basic idea is to evaluate an incoming tuple at once based on the index, rather than to evaluate it separately by each predicate (Fig-

* This work was supported by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korea government (MOST) (No. R0A-2007-000-10046-0).
The index helps to find a set of interval predicates efficiently that are matched by input tuples. Finding those matches over interval predicates is referred to as the *stabbing query problem* [5] in literature. There has been substantial research on predicate indexing methods to address this problem such as the CEI-based index [5,6], the Grouped filter [7,8], the PB+-tree [9], the interval binary search tree (IBS-tree) [4], the interval skip list (IS-list) [10], and so on.

The CEI-based index can be characterized by its fixed index structure. It assumes the range of input values is known in advance and decomposes the range into fixed-size segments. Each segment is again decomposed to non-overlapping smaller-size ones. A predicate interval is mapped to one or more consecutive segments. The CEI-based index achieves good search performance by arranging the segments properly. However, its fixed structure may degrade insertion performance significantly, especially when handling open intervals that can be distributed to a large number of segments.

Most of other indexes use balanced search trees which determine the segments dynamically based on endpoint values of predicate intervals. The grouped filter uses four separate structures to process greater-than, less-than, equality and inequality predicates, respectively. This separation facilitates insertion of predicates. But at the same time, it degrades search performance because we need to check up all indexes separately, and then merge matches retrieved from those indexes to organize a search result. It is also unclear how to deal with predicates with closed intervals in this method although they are commonly used in real-world applications.

The PB+-tree is based on the B+-tree where adjacent leaf nodes are connected by links. It has additional ordered lists that maintain greater-than, less-than, equality
and inequality predicates in order of their endpoint values, respectively. A leaf node has four slots, each of which is organized to point a predicate in each list. Despite of its simplicity, its insertion performance is relatively slow compared with other methods; its insertion cost is $O(n)$, where $n$ is the number of predicates in the tree. In addition, it is unclear how to process closed interval predicates.

The IBS-tree is a balanced binary search tree. Distinguished from the grouped filter and the PB+-tree, predicates can be placed in intermediate nodes in this method. This structure enables to process closed interval predicates gracefully (see Section 2). On the other hand, it leads to duplicate insertion of a new predicate, which consumes more memory. In addition, predicates in tree nodes need to be redistributed when tree balancing occurs. This process is quite complex, which requires $O(\log^2 n)$ insertion cost. The IS-list is proposed by the same author of the IBS-tree, and it is similar in principle to the IBS-tree while providing easier implementation. Their insertion and search performances are also similar, which is stressed in [10].

In this paper, we propose a hybrid indexing method to support fast insertion of predicates in data stream processing. We modify the structure of ordered lists in the PB+-tree scheme to reduce insertion cost to $O(\log n)$ (from $O(n)$ in the original scheme) and use the lists to process open interval predicates efficiently. The lists enable to reduce the number of predicates per tree node and avoid predicate redistribution related to the tree balancing. But, they are inappropriate to process closed interval predicates. To resolve this, we employ an algorithm of the IBS-tree to deal with closed interval ones.

As a result, our method provides better insertion performance than the IBS-tree when processing open interval predicates, while provides similar performance when processing closed interval ones. Our experimental results show that (1) the proposed method provides approximately 3 times better insertion performance when there are both of open and closed interval predicates with ratio of 1:1 and (2) the performance gap between two methods can be more than 12 times as the ratio of open interval predicates increases.

The remaining part of this paper is organized as follows. Section 2 reviews structures and algorithms of the PB+-tree and the IBS-tree. Section 3 proposes our method to improve insertion performance. Section 4 provides experimental results that compare insertion and search performances of the IBS-tree and our method.
Section 5 concludes our discussion.

2. BACKGROUNDS

In this section, we review the structures, insertion and search algorithms of the PB+-tree [9] and the IBS-tree [4] which we use in our method. Table 1 summarizes space and time complexities of the existing methods described above, including our method, where \( n \) is the number of predicates in the tree and \( L \) is the number of predicates in a search result. \( OIs \) and \( CIs \) stand for open and closed intervals, respectively.

<table>
<thead>
<tr>
<th>Method</th>
<th>Search cost</th>
<th>Insertion cost</th>
<th>Storage cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grouped filter</td>
<td>( O(\log(n + L)) ) for ( OIs ) * N/A for ( CIs )</td>
<td>( O(\text{N/A}) ) for ( OIs ) * N/A for ( CIs )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>PB+-tree</td>
<td>( O(\log(n + L)) ) for ( OIs ) * N/A for ( CIs )</td>
<td>( O(n) ) for ( OIs ) * N/A for ( CIs )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>IBS-tree</td>
<td>( O(\log(n + L)) )</td>
<td>( O(\log^2(n)) )</td>
<td>( O(n \log(n)) )</td>
</tr>
<tr>
<td>IS-list</td>
<td>( O(\log(n + L)) )</td>
<td>( O(\log^2(n)) )</td>
<td>( O(n \log(n)) )</td>
</tr>
<tr>
<td>Our method</td>
<td>( O(\log(n + L)) )</td>
<td>( O(\log(n)) ) for ( OIs ) * ( O(\log^2(n)) ) for ( CIs )</td>
<td>( O(n) ) for ( OIs ) * ( O(n \log(n)) ) for ( CIs )</td>
</tr>
</tbody>
</table>

\* We think the cost is \( O(n) \), which is discussed in Section 2.

2.1 PB+-tree

The PB+-tree is originally proposed as a disk-based index. It is based on the B+-tree structure where all leaf nodes are connected by links (Figure 2). Each node consists of four slots which point greater-than, less-than, equality and inequality predicates, respectively. Each type of predicates is maintained in an ordered list. In the figure, \( L_h \) and \( L_{gg} \) denote the lists that have less-than and greater-than predicates, respectively. Predicates in \( L_h \) are maintained in an increasing order of their endpoint values, while those in \( L_{gg} \) are kept in a decreasing order of the values. We only show less-than and greater-than slots and lists for simplicity. For convenience, we refer to these slots as the LT and GT slots, respectively.
Whenever a less-than predicate "P: x < v" is given, the PB+-tree adds new node v to the tree, and places P in \( L_{lt} \). Then, the LT slot of node v is organized to point P in \( L_{lt} \), and its GT slot is filled in with a value in the GT slot of its predecessor. For example, suppose a predicate "F: x < 17" is given to the tree in Figure 2. Then, the PB+-tree adds node 17 to the tree and places F after E in \( L_{lt} \). The LT slot of node v has a pointer to F, and its GT slot has a pointer to A which is the value in the GT slot of node 12 (i.e., node 17's predecessor). Since a new predicate is inserted to the tree without duplication, the storage complexity of this method becomes \( O(n) \).

The insertion of a less-than predicate gives influence to other predicates in \( L_{gt} \). In the above example, the LT slot of node 12 needs to be updated to point F, after the insertion of F. Otherwise, inaccurate search result can occur. If a tuple with value 12 arrives in this case, the result becomes a union of \{C\} and \{A, D\}, which are sublists of \( L_{lt} \) and \( L_{gt} \) starting from predicates pointed by the LT and GT slots of node 12. In this result, F is missed although its condition is actually satisfied with the input tuple. Note that the insertion of a new predicate may update all of existing predicates in the worst case. This leads to the insertion cost of \( O(n) \).

In the PB+-tree, it is unclear how to deal with closed interval predicates. One of the possible ways to deal with a closed interval predicate "P: a < x < b" is to decompose it into two parts "x > a" and "x < b", then add each part to the tree separately. When using this approach, P must be included in a search result only when it appears in both results retrieved from \( L_{lt} \) and \( L_{gt} \). However, in the worst case, all of predicates can be included in the result and we need to scan all of them to check if each one appears in both results. Due to this process, the search cost of the PB+-tree
becomes $O(n)$ in this case.

### 2.2 IBS-tree

The IBS-tree is a balanced binary search tree. Figure 3 (a) shows an example of the IBS-tree constructed from 5 predicates A to E which are the same as those in Figure 2. 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 less-than, equal to and greater-than the endpoint, respectively. For convenience, we refer to those slots as the LT, EQ and GT slots, respectively. We may indicate the three slots of node $v$ by using alternative notations $S_{lt}(v)$, $S_{eq}(v)$ and $S_{gt}(v)$.

Distinguished from the PB+-tree and the grouped filter, predicates can be placed in intermediate nodes in this method. This structure enables to process closed interval predicates gracefully. If we place predicates only in leaf nodes, a closed interval predicate "$a < x < b$" needs to be placed in every node between the interval $a$ and $b$, which leads to $O(n)$ insertion time in the worst case. On the other hand, if we allow them to be placed in intermediate nodes, the number of nodes in which we need to insert a given predicate can be limited to $\log n$. However, in this case, predicates in tree nodes need to be redistributed when tree balancing occurs, which is quite a complex process.

For example, consider the insertion of "$F: x \geq 2$" in the status of Figure 3 (a). The insertion of F incurs unbalance of the tree: the left subtree of node 12 outweighs its right subtree by the insertion of new node 2. To rebalance it, any of the existing
algorithms such as the AVL tree and the Red-black tree can be used. The tree rebal-
ance incurs redistribution of its predicates. In Figure 3 (a), predicates of node 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 described as follows. Below, \( f(S) \) denotes a function that returns a set of predicates in slot \( S \).

\[
\begin{align*}
    f(S_{\alpha}(\alpha')) &= f(S_{\alpha}(\alpha)) \cup f(S_{\beta}(\beta)) \\
    f(S_{\alpha}(\alpha')) &= f(S_{\alpha}(\alpha)) \cup f(S_{\beta}(\beta)) \\
    f(S_{\alpha}(\beta')) &= f(S_{\alpha}(\beta)) - f(S_{\alpha}(\alpha)) \\
    f(S_{\alpha}(\beta')) &= f(S_{\alpha}(\beta)) - f(S_{\alpha}(\alpha))
\end{align*}
\]

There are also redistribution rules for other cases (i.e., the cases of LR, RL and RR rotations). For complete rules, please refer to the original paper [4].

To insert a predicate with a closed interval "P: \( a < x < b \)" to the IBS-tree decom-
poses it into two parts "\( x > a \)" and "\( x < b \). Then, it adds a new node for each part to the tree. To handle the greater-than part "\( x > a \)", algorithm AddLeft (Figure 4) is used. (There is also a symmetric algorithm for the less-than part "\( x < b \).) In the algorithm, lval and rval are the left and the right endpoints of a predicate, and lineq denotes the inequality type of its greater-than part (i.e., "">
or "\( \geq \)"). The value, the left and the right subtrees of a node are denoted as value, lchild and rchild, respectively.

Function rightUp returns the lowest ancestor of the node \( N \) in the tree that con-
tains \( N \) in its left subtree. As shown in step (3) and (9), a closed interval can be in-
serted to \( S_{g}(N) \) only when rightUp(N) is smaller than or equal to its right endpoint.

---

**AddLeft(node \( N \), predicate \( P \))**

1. IF \( N = \text{null} \), create a node with value \( P.lval \) and assign it to \( N \) FI
2. IF \( N.val = P.lval \),
3. IF \( \text{rightUp}(N) \leq P.rval \), add \( P \) to \( S_{g}(N.val) \) FI
4. IF \( P.lineq = \text{"\( \geq \)"}, add \( P \) to \( S_{g}(N.val) \) FI
5. ELSE IF \( N.val < P.lval \),
6. AddLeft(\( N.rchild, P \))
7. ELSE,
8. IF \( N.val < P.rval \), add \( P \) to \( S_{g}(N.val) \) FI
9. IF \( \text{rightUp}(N) \leq P.rval \), add \( P \) to \( S_{g}(N.val) \) FI
10. AddLeft(\( N.lchild, P \))
11. FI

Fig. 4. Insertion of the left-end of a predicate to the IBS-tree
For example, the rightUp of node 7 is node 12 in Figure 3 (a). If a predicate "P: 6 < x < 11" is given to the tree, it is not added to $S_g(7)$ because step (9) is not satisfied. On the other hand, if P is given to "6 < x < 13", it is added to the slot. After step (11), tree rebalance and predicate redistribution are conducted as discussed above.

The storage complexity of the IBS-tree is $O(n \log n)$ because an open interval predicate can be added up to $\log n$ nodes duplicately. Its search cost is $O(\log n + L)$, where $n$ is the number of nodes in the tree and $L$ is the number of predicates in a search result. This follows since traversing a path from a root to a leaf in the tree requires $O(\log n)$ time using a balanced tree scheme and we spend $O(1)$ time to examine each of the predicates retrieved.

The cost of insertion in the IBS-tree is somewhat more difficult to calculate. It can be represented as a sum of three costs. The first is the cost of traversing the tree to find a leaf to add a new node for a given predicate, which is $O(\log n)$. The second is of rebalancing the tree when it is unbalanced by the new node, which is also $O(\log n)$ as discussed in [4]. The last one is of redistributing predicates of tree nodes when tree rebalancing occurs. The cost is $O(n^2)$ if predicates in each slot are not sorted, since conducting set difference or union operation requires $O(n^2)$ time in this case. Regarding this, the IBS-tree uses auxiliary binary search trees to keep the predicates in a sorted order. This reduces the redistribution cost to $O(n)$. But at the same time, it makes the first cost increase to $O(\log^2 n)$ because we need to insert a predicate to the auxiliary tree of a leaf node (after the tree traversing). Consequently, insertion cost of the IBS-tree becomes $O(\log^2 n)$.

3. OUR METHOD

Our method is based on a balanced binary search tree, which is identical to the IBS-tree. We utilize ordered lists proposed in the PB+-tree scheme to process open interval predicates more efficiently. To facilitate insertion of a predicate to the list, we keep links between adjacent tree nodes so that the tree can be manipulated as an ordered list. Figure 5 shows two examples of our search tree which are equivalent to those of Figure 3. A tree node has links to its inorder predecessor and successor, which are depicted as dashed lines with arrowheads. Similar to the PB+-tree, the LT and GT slots of a node point predicates in $L_l$ and $L_g$, respectively. (Equality pre-
3.1 Insertion algorithm

Let us first discuss the insertion of a new predicate in our search tree. When a less-than predicate "P: x < v" is given, we newly add node v to the tree and insert P to the list \( L_{lt} \). To keep the order of \( L_{lt} \), we place it after a predicate of node v's predecessor. After the insertion of P, we set the LT slot of node v to point P in \( L_{lt} \) and fill in the GT slot with the value in the GT slot of the predecessor. For example, if a predicate "G: x < 9" is given in Figure 5 (b), we add node 9 to the left-hand side of node 10, and insert G after B which is the predicate of node 9's predecessor (Figure 6). Then, we set the LT and GT slots of node 9 to point G and D, respectively.

The predecessor may be originated (i.e., generated) from a greater-than predicate, not a less-than one. In this case, the order of \( L_{lt} \) cannot be preserved if we place P after a predicate of the predecessor. Suppose a predicate "H: x < 17" is given after the insertion of G. Then, we add node 17 to the left-hand side of node 20 and find its predecessor to add H to \( L_{lt} \). The predecessor of node 17 is node 12 which is originated from a greater-than predicate A. If we add H after node 12's predicate C, the order of predicates in \( L_{lt} \) is broken.
This problem can be resolved by inserting P after another nearest predecessor that has the same origin (i.e., that is originated from a less-than predicate). Let us call tree nodes originated from less-than and greater-than predicates simply \( LT\)-origin and \( GT\)-origin nodes, respectively. Then, we can see that the predecessor (or the successor) relationship is valid among tree nodes only of the same origin, regarding the insertion of predicates. For convenience, we refer to the predecessor (or the successor) that has the same origin of node \( v \) as a valid predecessor (or a valid successor) of node \( v \). In the above example, the valid predecessor of node 17 is node 10, so we add H after its predicate E.

As discussed in the review of the PB+-tree scheme, the insertion of a less-than predicate can give influence to other predicates in \( L_{gt} \). In the above example, the LT slot of node 12 needs to be updated to \( H' \) from \( C' \), after the insertion of H. This updating process may need to scan all of tree nodes in the worst case. To avoid the exhaustive scan, we divide our search tree into a number of groups, each of which includes adjacent tree nodes with the same origin. We refer to a group of LT-origin (or GT-origin) nodes as a LT-origin (or GT-origin) group. In Figure 6, there are two LT-origin groups \{node 7, node 9, node 10\} and \{node 17, node 20\}, and two GT-origin groups \{node 2, node 3\} and \{node 12\}. 

Fig. 6. Insertion of "G: \( x < 9 \)" and "H: \( x < 17 \)" in the tree of Figure 5 (b)
In the insertion of the less-than predicate $P$, we need to update the LT slots of all nodes in a GT-origin group of a predecessor of node $v$. Note that the LT slots of all nodes in the group have the same value in the LT slot of node $v$ (e.g., consider the insertion of $H$). Since the member nodes always have the same value of $S_{lt}(v)$, it is possible to use a common slot (i.e., variable) to share the value. Then, by updating the value through the slot at once, we can avoid repetitive updates in every LT slot of member nodes. For this purpose, we maintain a common slot for each group which we call a group slot. The slot of a LT-origin group is pointed by the GT slots of its member nodes, while the slot of a GT-original one is shared by the ST slots of its members.

Figure 7 shows an algorithm for the insertion of a less-than predicate in our method. It describes the process after a tree node $N$ is created for the given predicate $P$. We first find a valid predecessor $M$ of $N$ and add $P$ after a predicate of $M$ in $L_{lt}$. Then, we set $S_{lt}(N.val)$ to point $P$, where we use "$\leftarrow"$ to denote assignment operation. If $P$ involves an equality check, we also add it to $S_{eq}(N.val)$. Since $N$ is a LT-origin node, its GT slot $S_{gt}(N.val)$ is organized to point a group slot $G.slot$, where $G$ is a group of $N$. If $G$ does not exist, we create it and initiate its slot with the value of $S_{gt}(M.val)$. Finally, we update the group slot of $M$, i.e., assign $P'$ to $F.slot$, where $F$ is the group of $M$. The algorithm for insertion of a greater-than predicate is symmetric to this, and we omit its discussion from the lack of spaces.

Note that in our method, there is no predicate redistribution when rebalancing.
the tree. Predicates in $L_{lt}$ are maintained in an increasing order of node values, which order is exactly the same as the order obtained by inorder traversal of the tree. One of the intrinsic characteristics of binary search trees is that the traversal order is not broken after the tree is rebalanced. From this, we can see that the predicate order of $L_{lt}$ is not influenced by tree rebalancing (as we can see in Figure 5 (a) and (b)). Similarly, the predicate order of $L_{gt}$ is also observed in the same situation.

Insertion cost of the method described above is $O(\log n)$ which is required to traverse the tree to find a leaf where a new node for a given predicate is added to. Finding a valid predecessor or successor can be conducted in $O(1)$ if we organize $L_{lt}$ and $L_{gt}$ to be doubly-linked lists whose elements also have pointers to their corresponding tree nodes. Other processing relevant to group slots costs $O(1)$. Most of all, there is no predicate redistribution in this method which takes additional $O(\log n)$ time.

### 3.2 Search algorithm

Now, let us discuss the search algorithm of our method. Whenever a new tuple arrives, we first identify a node where the search ends. Then, we organize a search result based on the values of three slots of the end node. Basically, we union the predicates in its EQ slot and two sublists of $L_{lt}$ and $L_{gt}$ starting from predicates pointed

---

**Search**

**input V**

1. Find a node $N$ where the search ends
2. $G \leftarrow$ a group of $N$
3. IF $N$ is a LT-origin one,
   4. IF $V > N.val$,
      5. $M \leftarrow$ a valid successor of $N$
      6. Return $f(S_{eq}(N.val)) \cup g(L_{ls}, S_{eq}(M.val)) \cup g(L_{gt}, G.slot)$
   7. ELSE,
      8. Return $f(S_{eq}(N.val)) \cup g(L_{ls}, S_{eq}(N.val)) \cup g(L_{gt}, G.slot)$
   9. IF
10. ELSE IF $N$ is a GT-origin one,
11. IF $V < N.val$,
12. $M \leftarrow$ a valid predecessor of $N$
13. Return $f(S_{eq}(N.val)) \cup g(L_{ls}, G.slot) \cup g(L_{lt}, S_{eq}(M.val))$
14. ELSE,
15. Return $f(S_{eq}(N.val)) \cup g(L_{ls}, G.slot) \cup g(L_{lt}, S_{eq}(M.val))$
16. IF
17. FI

Fig. 8. Organization of a search result in our method
by its LT and GT slots. We slightly vary the organization of the result according to the end node type (i.e., origin) and a comparison result of an input tuple.

Let the end node have value $v$. Then, if the node is a LT-origin one, we need to exclude a predicate pointed by its LT slot from the result when an input value is larger than $v$. For example, when a tuple with value 11 arrives in Figure 6, the search ends in node 10, but its predicate $E$ must be excluded from a result because it is defined as "$x < 10\)". To do this, we find a valid successor of node $n$ and return a sublist starting from a predicate pointed by the successor's LT slot in this case. In the opposite case where the node is a GT-origin one, we find a valid predecessor and return a sublist retrieved from the value of its GT slot.

Figure 8 shows the search algorithm of our method. We first identify the end node $N$ and its group $G$. Then, if $N$ is a LT-origin one and the input value $V$ is larger than $N.val$, we exclude a predicate pointed by $S_l(N.val)$ from the union result. Otherwise, we return the union without a change. In the algorithm, $f(S)$ is a function that returns the predicates of slot $S$ and $g(L, S)$ is a function that returns a sublist of $L$ starting from a predicate pointed by $S$. The processing for the case when $N$ is a GT-origin one is symmetric to the above description, which is described in step (10) to (17).

Time complexity of our search algorithm is the same as that of the IBS-tree - $O(\log n + L)$. It takes $O(\log n)$ to traverse the tree to find $N$ (step (1)). All other steps are just a constant factor which is $O(1)$. There is other post-processing to examine the retrieved results which takes $O(L)$ time, where $L$ is the number of retrieved predicates; we do not describe the post process in Figure 8.

3.3 Closed interval handling

Note that the method described above can properly work only when all of the given predicates are those with open intervals. To handle a closed interval predicate "$a < x < b\)$", we may decompose it into two parts "$x > a\)$ and "$x < b\)$", then add each part to the tree separately. But, this approach can degrade the search performance significantly because we may have to scan all of predicates in the worst case, as discussed in Section 2. Therefore, we employ the insertion algorithm of the IBS-tree $AddLeft$ to deal with closed interval predicates. In other words, we use a hybrid method based on the IBS-tree and our algorithms: the former for closed interval predi-
cates and the latter for open interval ones.

Compared with the IBS-tree, our method has more things to do in both of the insertion and the search. For the insertion, we need to initiate the LT and GT slots of two new tree nodes generated from a closed interval predicate. The initiation will be placed after step (2) in AddLeft. For the search, we need to gather predicates of the nodes in a search path from a root to a leaf before conducting our search algorithm, since intermediate nodes may have predicates in its slots resulting from AddLeft. These augmented parts incur additional cost when handling closed interval predicates. However, the cost is insignificant, which we observed through our experiments.

Time complexity of our search algorithm in this case is the same as that of the IBS-tree - $O(\log n + L)$. Actually, our method conduct 4 or 5 more steps than the IBS-tree to return sublists of $L_{lt}$ and $L_{gt}$, but these are just a constant factor that does not give influence to the algorithm's complexity. Insertion cost of our method is definitely the same as that of the IBS-tree when handling closed interval predicates - $O(\log^2 n)$, while it is different from it when dealing with open interval predicates – $O(\log n)$ as discussed in Section 3.1.

4. EXPERIMENTAL RESULTS

In this section, we provide experimental results that compare the performances of the IBS-tree and our method. We did not compare with other methods mentioned above such as the PB+tree and the Grouped filter, because it is not clear how to process closed interval predicates in those methods (although such predicates are commonly used in real-world applications).

For our experiments, we implemented algorithms of both methods as well as a data generator to synthesize test data sets. We set the range of predicate values to [0, 1000000] and randomly produced up to 128,000 predicates with random inequality types (i.e., "<", "\leq", ">" and "\geq"). Three kinds of data sets were used to simulate the worst, the general and the best cases of our method, respectively. The worst-case data set has only closed interval predicates, while the best-case data set has only predicates with open intervals. The general-case data set has both of open and closed interval predicates, and in this case, we organize the ratio of two kinds of predicates.
to be the same. Our experiments were conducted on Intel Pentium IV 2.4 GHz machine, running Window XP, with 1G main memory.

First, we would compare search performances of two methods. We increased the number of predicates from 8K to 128K and conducted 100,000 times of searches for each data set. Figure 9 (a) shows the search times of two methods. We provided only one result obtained from the worst-case data set, since the results from other data sets are similar to it. As shown in the figure, search performances of two methods are almost the same although our method performs 4 or 5 more steps to process an input tuple. We observed that their performances are the same even when we decrease the number of predicates to 100, which result is not provided here. These results show that the influence of our augmented codes is insignificant, compared with two dominant factors which include traversing a tree from a root to a leaf and examining each of retrieved predicates.

Then, we compared insertion performances of two methods. We increased the number of predicates from 8K to 128K and observed their insertion times for each
data set. Figure 9 (b), (c) and (d) show the insertion times for the worst, the general and the best cases, respectively. In the worst case, the IBS-tree is better than our method, but their performance gap is less than 1% (actually, less than 0.2%) which is hardly identified in the figure. In other cases, our method provides better insertion performance than the IBS-tree, and their gap increases as the ratio of open interval predicates becomes larger. For each general-case and best-case data set, our method provides approximately 3 times and 12 times better performance than the IBS-tree, respectively.

5. CONCLUSION

In this paper, we proposed a hybrid indexing method to support fast insertion of predicates in processing of continuous data streams. We modified the structure of ordered list proposed in the PB+-tree scheme and applied those lists to process open interval predicates more efficiently. We used an algorithm of the IBS-tree to handle closed interval predicates. As a result, our method provides better insertion performance than the IBS-tree when dealing with open interval predicates, while provides similar performance in the other case. Regarding this, our method requires $O(\log n)$ insertion time for the former case, while requires $O(\log^2 n)$ for the latter case which is the same as that of the IBS-tree. Through our experiments, we observed that (1) the proposed method provided approximately 3 times better insertion performance than the IBS-tree in a general situation where there are both of open and closed interval predicates and (2) their performance gap was up to more than 12 times when there are only open interval predicates.

REFERENCES

3. L. Golab and M. T. Ozsu, “Issues in Data Stream Management”, *ACM SIG-
MOD Record, Vol.32 (2), 2003, pp.5-14.


Hyeon Gyu Kim (金賢奎) received his B.S and M.S. degrees in computer science from University of Ulsan, Ulsan, South Korea, in 1997 and 2000, respectively. He has been a researcher in LG Electronics, Seoul, South Korea, from March 2000 to February 2005. He is currently pursuing his Ph.D. degree in Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea. His research interests include database systems and data stream processing.
Myoung Ho Kim (金命鎬) received his B.S. and M.S. degrees in Computer Engineering from Seoul National University, Seoul, Korea in 1982 and 1984, respectively, and received his Ph.D. degree in Computer Science from Michigan State University, East Lansing, MI, in 1989. He joined the faculty of the Department of Computer Science at KAIST, Daejeon, Korea in 1989 where currently he is a professor. His research interests include database systems, data stream processing, sensor networks, mobile computing, OLAP, XML, information retrieval, workflow and distributed processing.