A Hybrid Method for Fast Predicate Matching in Data Stream Processing

HYEON GYU KIM AND MYOUNG HO KIM
Department of Computer Science
Korea Advanced Institute of Science and Technology
Daejeon, 305-701 Korea

Efficient predicate matching is essential to provide real-time responses in data stream processing. Many predicate indexes have been introduced to address the predicate matching problem. Among them, the IBS-tree handles predicates with open and closed intervals in $O(\log n)$ time. However, it was found that its insertion algorithm is not optimal when processing open intervals. In this paper, a predicate indexing method is proposed that supports fast insertion of predicates based on the IBS-tree scheme without degrading the search performance. The basic idea of the proposed method is to use shared lists when processing open intervals, and the insertion time is reduced to $O(\log n)$ in this case. When dealing with closed intervals, an algorithm of the IBS-tree was used. Consequently, the insertion performance of the proposed method improves as the ratio of open intervals increases, which we observed through our experiments.

Keywords: predicate matching, predicate index, shared list, IBS-tree, data stream processing

1. INTRODUCTION

Recent years have witnessed the emergence of a new class of applications that monitor continuous streams of data items such as auction bids, stock exchanges, network measurements, sensor readings, and so on [1-3]. One popular application of stream monitoring is to find predicate intervals matching an input event [4, 5]. For example, in a stock trading application, users may pose queries with interval predicates (e.g. $x < v$, $x > v$ or $x = v$, where $x$ and $v$ are a variable and a constant, respectively) to monitor real-time changes in stock prices. Whenever a price change is detected in this application, the system will find predicates whose intervals satisfy the given price, and notify users of the matching predicates.

In general, a predicate index is used to find those matches efficiently in large numbers of predicate intervals. The index is commonly in the form of a binary search tree as shown in Fig. 1, whose nodes correspond to distinct endpoint values of the predicate intervals. It enables a reduction in the search time from $O(n)$ to $O(\log n)$ where $n$ is the number of distinct predicate intervals.

There have been many predicate indexes introduced so far, including 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. It is important to note that their search and insertion performances differ slightly according to the types of predicate intervals that they handle.
(1) open intervals \((x < v, x > v \text{ or } x = v)\) and (2) closed intervals \((v_1 < x < v_2, \text{ where } v_1 \text{ and } v_2 \text{ are constants})\). The costs associated with these indexes are summarized in Table 1, where \(OI\) and \(CI\) stand for the open and closed intervals, respectively. For the Grouped filter and the PB+-tree, it is unclear how these indexes handle closed intervals, so N/A is denoted for those cases in the table. The algorithms and costs of the existing methods are discussed further in section 2.

In this paper, a new predicate indexing method is proposed that supports fast predicate insertion while providing similar search performance to existing methods. The proposed method is based on the IBS-tree which provides an \(O(\log_2 n)\) insertion time and \(O(\log n + L)\) search time, where \(L\) is the number of predicates in the search result. The index’s insertion and search algorithms are used to handle closed intervals. However, it was found that the IBS-tree’s insertion algorithm is not optimal when processing open intervals. To address this, a shared list whose structure is similar to that of the PB+-tree was employed. Using this structure, the insertion cost was reduced to \(O(\log n)\) when processing open intervals.

The remainder of this paper is organized as follows. Section 2 reviews the structures and algorithms of the existing methods. Section 3 proposes the new method to improve the insertion performance. Section 4 provides experimental results which compare the proposed method with the existing algorithms in terms of insertion and search costs. Section 5 concludes the discussion.

### 2. BACKGROUND

In this section, the structures and algorithms of the existing methods are reviewed.
A HYBRID METHOD FOR FAST PREDICATE MATCHING

A: $x \geq 12$
B: $x < 7$
C: $x < 20$
D: $x \geq 3$
E: $x < 10$

Slot Li
Slot Gi

First, the PB+-tree which uses ordered lists to share predicates among tree nodes is introduced. Then, the IBS-tree which deals with closed intervals well in $O(\log^2 n)$ time is discussed. Finally, the other methods including the grouped filter, the IS-list and the CEI-based index are summarized.

2.1 PB+-tree

The PB+-tree [9] is based on the B+-tree structure where all leaf nodes are connected by links. Fig. 2 shows an example of the PB+-tree constructed from predicates A to E. For each distinct endpoint of the predicate intervals, there is a leaf node consisting of three slots, each of which has a set of predicates whose ranges are less than, equal to, and greater than the value, respectively. Those slots of the node $N_i$ (i.e. a node whose value is $i$) are denoted as $L_i$, $E_i$, and $G_i$, respectively. (Here, the equality slots $E_i$ are omitted to simplify the discussion.)

In the PB+-tree, each slot does not have a full set of predicates whose intervals overlap a node value. Instead, the PB+-tree stores predicates in a shared list and makes a slot to point to a predicate in the list. There are two types of lists in the PB+-tree: (1) a list of less than predicates $OL$ and (2) a list of greater than predicates $OG$. Predicates in the list $OL$ are maintained in an increasing order of their endpoint values, while those in the list $OG$ are in a decreasing order of the values. A slot $L_i$ (or $G_i$) has a pointer to an element in $OL$ (or $OG$). In this scheme, the actual value of a slot can be obtained by retrieving a list of the consecutive elements starting from one indicated by the slot. For example, $S(L_{10}) = \{E, C\}$ and $S(G_{10}) = \{D\}$ in the above figure, where $S(x)$ is a function that returns a set of predicates designated by a slot $x$.

The insertion algorithm of the PB+-tree is intuitive. Suppose that a less-than predicate $F: x < 17$ is given in Fig. 2. Then, the PB+-tree adds a new node $N_{17}$ and inserts $F$ in $OL$. When inserting a predicate, it may check multiple predecessors of the new node. This is because the predecessors of the node may indicate predicates whose endpoints are larger than a given predicate. In the above example, $N_{12}$ points $C$ whose endpoint is 20 (larger than 17). Thus, the PB+-tree moves to the next predecessor $N_{10}$ and inserts $F$ after its predicate $E$. Then, it configures $L_{12}$ as well as $L_{17}$ to point to the new predicate. The organization of $G_{17}$ is symmetric to that of $L_{17}$.

Note that the PB+-tree may scan and update all tree nodes to insert a new predicate.
without violating the order of $O_l$ or $O_r$. This leads to an insertion time of $O(n)$. In addition, it is unclear how to deal with closed intervals in the PB+-tree scheme. In the next subsection, the IBS-tree which processes closed intervals in $O(\log^2 n)$ time is introduced.

### 2.2 IBS-tree

The IBS-tree [4] is a balanced binary search tree which extends to support interval handling. Fig. 3 (a) shows an example of the IBS-tree constructed from the same predicates $A$ to $E$ used in Fig. 2. A tree node consists of three slots $L_i$, $E_i$, and $G_i$, whose meanings are the same as those in the PB+-tree.

![IBS-tree constructed from predicates $A$ to $E$.](image)

To illustrate the insertion algorithm of the IBS-tree, consider the case when $F: x \geq 2$ is given in Fig. 3 (a). The IBS-tree first adds a new node $N_2$ to the tree and inserts $F$ to slots $G_2$ and $E_2$. During the addition, the predicate $F$ is also inserted into the $G_i$ and $E_i$ of all nodes $N_i$ which are inorder successors of $N_2$ and at the same time are its ancestors. After the insertion, the IBS-tree starts the LL rotation of the AVL-tree scheme to balance the tree, since the left subtree of root $N_{12}$ outweighs the right one. Then, it redistributes the predicates of $N_\alpha$ and $N_\beta$ whose statuses are changed by the rebalancing. Let $S(x)$ be a function to return a set of predicates in slot $x$. Then, the redistribution rule can be described as follows,

\[
\begin{align*}
S(L_\alpha') &= S(L_\alpha) \cup S(L_\beta); \\
S(E_\alpha') &= S(E_\alpha) \cup S(L_\beta); \\
S(G_\beta') &= S(G_\beta) - S(G_\alpha); \\
S(E_\beta') &= S(E_\beta) - S(G_\alpha).
\end{align*}
\]

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

The insertion of a predicate with a closed interval $a < x < b$ can be easily completed by separately inserting two parts, $x > a$ and $x < b$, which are decomposed from the original interval. When inserting each part, the IBS-tree conducts an additional check using the method rightUp or leftUp. The rightUp($N_i$) method returns the lowest ancestor of $N_i$.
from the tree that contains $N_i$ as its left subtree [4]. The method is used to insert the greater than part $x > a$. The IBS-tree inserts the predicate to $G_i$ and $E_i$ only when the value of the node returned by rightUp($N_i$) is smaller than or equal to the right endpoint $b$. For example, rightUp(7) is $N_{i2}$ in Fig. 3 (a). If an interval $6 < x < 11$ is given, it is not added to $N_i$ because the node value 12 is larger than the right endpoint of 11. If an interval $6 < x < 13$ is given, it will be added. The method leftUp is used for an insertion in the less than part.

The search cost of the IBS-tree is $O(\log n + L)$, where $n$ is the number of nodes in the tree and $L$ is the number of predicates in the search result. The cost is because $O(\log n)$ time is required to find the search tree and $O(1)$ time is required to examine each predicate retrieved. The insertion cost of the IBS-tree is $O(\log^2 n)$. The insertion of a predicate into tree nodes costs $O(\log n)$ time. When inserting the predicate to each slot, an additional $O(\log n)$ time is required to maintain the order of predicate IDs [4], which is required to facilitate the predicate redistribution. These lead to an insertion time of $O(\log^2 n)$.

### 2.3 Other Methods

The grouped filter [7, 8] is a predicate index introduced in TelegraphCQ, which is a well-known stream processing engine. It consists of four structures including two balanced binary search trees and two hash tables. The search trees are used to handle the less than and greater than predicates, while the hash tables are for equality and strict inequality (i.e. “!=” predicates. Since the less than and greater than predicates are maintained in separate trees, there is no case when a predecessor (or a successor) of a new node points to a predicate whose endpoint is larger (or smaller) than the given predicate (see section 2.1). This leads to an insertion time of $O(\log n)$. However, the separate structure degrades the search performance at the same time. This is because all structures need to be checked, and then the search results returned are merged individually. In addition, it is not clear how to the closed intervals are handled in this method.

The interval skip list (IS-list) [10] is a skip list which has been extended to handle the predicate intervals. It is proposed by the same author of the IBS-tree and is similar in principle to the IBS-tree while providing easier implementation. The insertion and search performance of the IS-list and the IBS-tree are also similar, which is stressed in [10].

The CEI-based index [5, 6] 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 sized segments. 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 the insertion performance significantly, especially when working with open intervals that can be distributed to a large number of segments.

### 3. THE PROPOSED METHOD

As mentioned earlier, the proposed method is based on the IBS-tree and utilizes its algorithm to insert closed intervals in $O(\log^2 n)$ time. However, it was found that the al-
algorithm is not optimal when processing open intervals. For instance, when a predicate \( x \geq v \) is inserted into the slots \( G_i \) and \( E_i \) of all nodes \( N_i \) which are inorder successors of the new node \( N_v \) and at the same time are also its ancestors, as well as those slots of \( N_v \) (e.g. consider the insertion of \( F: x \geq 2 \) in section 2.2). Such duplicate insertion consumes more space as well as computation power. In this section, a method for the insertion of open intervals in \( O(\log n) \) time with less storage based on the IBS-tree is proposed.

![Figure 4. Examples of the proposed search tree.](image)

The basic concept of the proposed method is to use the shared lists introduced in the PB+-tree when processing open intervals. Fig. 4 shows examples of the proposed index structure which are equivalent to those of the IBS-tree in Fig. 3. There are two structural differences between the proposed method and the IBS-tree as follows,

1. The proposed search tree has shared lists \( O_l \) and \( O_g \) to store the less than and greater than predicates, respectively.
2. Each node in the proposed tree has two slots, \( \Lambda_i \) and \( \Gamma_i \) to store pointers to \( O_l \) and \( O_g \), respectively, as well as three slots of the IBS-tree (i.e. \( L_i \), \( E_i \), and \( G_i \)). (\( \Lambda_i \) and \( \Gamma_i \) are respectively equivalent to \( L_i \) and \( G_i \) in the PB+-tree.)

In the above figure, the slots \( L_i \), \( E_i \), and \( G_i \) are not depicted since the focus here is on how to handle open interval predicates; in the proposed method, the three slots are only used to store closed interval predicates where the open interval predicates are maintained in \( O_l \) or \( O_g \). When a value of \( \Lambda_i \) or \( \Gamma_i \) is referred to, \( P' \) is used to denote a pointer to a predicate \( P \) in the list. The following discussion looks at (1) how to set the values of \( \Lambda_i \) and \( \Gamma_i \) when inserting a new predicate and (2) how to obtain a search result from the values of \( \Lambda_i \) and \( \Gamma_i \).

Given a less than predicate \( P: x < v \), a new node \( N_v \) should be first added to the tree. Then, \( P \) is inserted into \( O_l \) and \( \Lambda_i \) is set to \( P' \). In the proposed method, \( O_l \) and \( O_g \) are implemented as threaded binary trees which support inorder traversal. Using the trees, a predicate can be inserted in \( O(\log n) \) time, without scanning inorder predecessors to find
the correct point whose predicate endpoint is smaller than \( v \) (see section 2.1).

After that, \( \Lambda_i \) (\( i < v \)) of the inorder predecessors of \( N_v \) are updated to point to the new predicate. As distinguished from the PB+-tree, not all inorder predecessors are updated. In the proposed method, only the ancestors of \( N_v \) among the predecessors are updated. For example, consider a predicate \( G: x < 5 \) is given in Fig. 4 (b). Then, the PB+-tree will update all predecessors to point \( G \) after inserting the predicate (i.e. set \( \Lambda_2 \) and \( \Lambda_3 \) to \( G' \)). However, the proposed method will only update \( N_3 \) which is an inorder predecessor of the new node \( N_v \) and at the same time it is its ancestor (i.e. set \( \Lambda_3 \) to \( G' \)).

Let \( f(x) \) be a function that returns an endpoint value of a predicate pointed to by a slot \( x \). Then, the update algorithm of the proposed method can be described as Fig. 5.

**Algorithm 1** \( \text{UpdateLambda}(P) \)

**Input:** a less than predicate \( P: x < v \),
1. Set \( \Lambda_v \) to \( P' \) and \( \Gamma_v \) to null
2. For all \( N_i \) (\( i < v \)) which are ancestors of \( N_v \)
3. If \( f(\Lambda_i) > v \), then set \( \Lambda_i \) to \( P' \)

**Algorithm 2** \( \text{UpdateGamma}(P) \)

**Input:** a greater than predicate \( P: x > v \),
1. Set \( \Gamma_v \) to \( P' \) and \( \Lambda_v \) to null
2. For all \( N_i \) (\( i > v \)) which are ancestors of \( N_v \)
3. If \( f(\Gamma_i) < v \), then set \( \Gamma_i \) to \( P' \)

**Fig. 5.** Updating \( \Lambda_i \) and \( \Gamma_i \) when inserting a predicate in the proposed method.

In the algorithm \( \text{UpdateLambda} \), steps 2 and 3 guarantee that the slot \( \Lambda_i \) of node \( N_i \) has a predicate whose endpoint value is always the smallest among the predicates of the right descendents of \( N_i \). Whenever a new node \( N_j \) (generated from a less-than predicate \( x < j \)) is added to the right subtree of \( N_i \), we check whether \( j \) is smaller than \( f(\Lambda_i) \). If so, \( f(\Lambda_i) \) is updated to have the value of \( \Lambda_j \). For example, if a predicate \( H: x < 15 \) is given in Fig. 4 (b), \( \Lambda_{12} \) will be updated to point the newly inserted predicate \( H \), so that \( f(\Lambda_{12}) \) will be the smallest among the values of other nodes in the right subtree of \( N_{12} \).

In a similar way, steps 2 and 3 in \( \text{UpdateGamma} \) ensure that the slot \( \Gamma_i \) of a node \( N_i \) has a predicate whose endpoint value is always the largest among the predicates of the left descendents of \( N_i \). From the above discussion, the following property of the proposed search tree has been derived. The property will be used to guarantee the accuracy of search results in the proposed method, which will be discussed further below.

**Property 1** If \( \Lambda_i \) is not null, \( f(\Lambda_i) \) is the smallest among \( f(\Lambda_j) \) where \( N_j \) is a right descendent of \( N_i \). Symmetrically, if \( \Lambda_i \) is not null, \( f(\Gamma_i) \) is the largest among \( f(\Gamma_j) \) where \( N_j \) is a left descendent of \( N_i \).

Note that the values of \( \Lambda_i \) and \( \Gamma_i \) are not changed when balancing the search tree. The lists \( O_L \) and \( O_G \) are also not influenced by the incident in the proposed method. This is because the relationship between a node and its inorder predecessor or successor is not changed when the tree structure is reorganized using the balancing scheme. From this, the predicate orders of \( O_L \) and \( O_G \) remain unchanged, and as a result, \( \Lambda_i \) and \( \Gamma_i \) do not
need to be updated when balancing the tree. This enables the proposed method to show good performance when inserting open intervals, which is confirmed by the experiments presented in section 4.

Now, let us discuss how to obtain search results from the values of $\Lambda_i$ and $\Gamma_i$. In the proposed method, if a search ends in a node $N_i$, $S(\Lambda_i)$ and $S(\Gamma_i)$ are always a super set of the search result. This is because, by definition, $S(\Lambda_i)$ and $S(\Gamma_i)$ are the sets of all predicates whose ranges are less than and greater than the endpoint value $i$, respectively. (As discussed earlier, the set can be obtained by retrieving a list of consecutive elements in $O_L$ or $O_G$ starting from a predicate pointed out by slot $\Lambda_i$ or $\Gamma_i$.) Therefore, a search result is generated based on $S(\Lambda_i)$ and $S(\Gamma_i)$ where $N_i$ is the last visited node included in the search path.

There are some exceptional cases that need to be handled when generating the search result. If the last node $N_j$ is generated from a less-than predicate and an input value $v$ is larger than $i$, the predicates with endpoint value $i$ need to be excluded from the result. For example, consider that an input event with a value of 11 arrives in Fig. 4 (b). Then, a search will end in $N_{i10}$ and $S(\Lambda_{i10})$ will be $\{E, C\}$. However, the predicate $E$ must not be included in the search result because its interval $x < 10$ does not overlap with the input value. In a similar way, predicates with endpoint values $i$ are also excluded when $N_i$ is generated from a greater than predicate and the input value $v$ is smaller than $i$. The predicate exclusion can be conducted without performance degradation: it can be accomplished by returning a list of elements in $O_L$ or $O_G$ starting from the predicate next to that pointed out by slot $\Lambda_i$ or $\Gamma_i$.

Note that $\Lambda_i$ or $\Gamma_i$ can be null in the proposed method. If $\Lambda_i$ is null, we use the value of another slot $\Lambda_j$ of a node $N_j$ which is an inorder successor of $N_i$ and at the same time is its ancestor. This is due to the Property 1 stating that $\Lambda_j$ indicates a predicate with the smallest endpoint value among those of the right descendants (also inorder successors) of $N_i$. In other words, the Property 1 guarantees that $S(\Lambda_j)$ is the largest set of predicates that overlap the input value. If $\Lambda_j$ is also null, the choice is moved to another node $N_k (k > j)$ which is an ancestor of $N_i$. This is continued until a non-null slot is found or until there are no more nodes to go through. If $\Gamma_i$ is null, the value of $\Gamma_j$ of node $N_j$ is used, which is an inorder predecessor of $N_i$ and at the same time is its ancestor.

For example, consider that an input event with a value of 1 arrives in Fig. 4 (b). Then, a search ends in $N_2$ and a search result is generated based on $S(\Lambda_2)$ and $S(\Gamma_2)$. Since $\Lambda_2$ is null, $\Lambda_3$ whose node is an ancestor of $N_2$ is checked. Again, $\Lambda_3$ is null, so the process moves to $N_7$ and $\Lambda_7$ is used to generate the search result. In addition, the first element in $S(\Gamma_2)$ is excluded and there is an empty set for it, since $N_2$ is generated from a greater-than predicate. After all, $\{B, E, C\}$ are returned as the search results.

To find a non-null slot, the search path does not need to be revisited. It can be identified during the search process. Let the last non-null slots included in the search path be $\Lambda_{i12}$ and $\Gamma_{i12}$. Then, for each node $N_i$ from a root to a leaf in the search path, $\Lambda_{i12}$ is set to $\Lambda_i$ if the node value $i$ is smaller than the input value $v$ and $\Lambda_i$ is not null. If $i$ is larger than $v$ and $\Gamma_i$ is not null, $\Gamma_12$ is set to $\Gamma_i$. From the above discussion, the proposed search algorithm can be summarized as Fig. 6.

It is clear that the search cost of the proposed method is $O(\log n)$ since a result can be identified in a single pass of the tree search. Compared with the IBS-tree, the proposed method conducts additional processing to obtain the predicates from $\Lambda_i$ and $\Gamma_i$, which is
Algorithm 3  Search($N_i$, $N_j$, $v$)
Input: node $N_i$, parent $N_j$, input value $v$
Output: a set of predicates
1. If $i = v$, then set $N_j$ to $N_i$ and go to step 3
2. If $N_j$ is null, then
3.  If $N_i$ is generated from a less than predicate, then
4.    If $j > v$, then exclude predicates with endpoint $j$ from $S(\Lambda_j)$
5.    Return $S(\Lambda_j)$ union $S(\Gamma_j)$
6.  Else if $N_j$ is generated from a greater than predicate, then
7.    If $j < v$, then exclude predicates with endpoint $j$ from $S(\Gamma_j)$
8.    Return $S(\Lambda_j)$ union $S(\Gamma_j)$
9.  End if
10. End if
11. If $i < v$ then
12.   If $\Lambda_i$ is not null, then set $\Lambda_j$ to $\Lambda_i$
13.   Return Search(a left child of $N_i$, $N_j$, $v$)
14. Else
15.   If $\Gamma_i$ is not null, then set $\Gamma_j$ to $\Gamma_i$
16.   Return Search(a right child of $N_i$, $N_j$, $v$)
17. End if

Fig. 6. Search algorithm of the proposed method.

described in the above algorithm. (The algorithm also has steps to obtain closed interval predicates from $L_i$, $E_i$, and $G_i$, which are omitted here to simplify the discussion.) However, its cost is not significant compared with the cost of evaluating each predicate in the result (e.g., checking if it is valid and copying it to a result list). The experimental results regarding this are presented in the next section.

4. EXPERIMENTAL RESULTS

In this section, we provide experimental results that compare the insertion and search performances of the PB+-tree, the IBS-tree and the proposed method.

4.1 Experimental Setup

To conduct the experiments, the above three algorithms were implemented. In case of the PB+-tree, it is not clear how to process closed intervals. Regarding this, there are two alternatives to implement it. One is to insert a new predicate to all tree nodes whose values are in the predicate interval $a < x < b$. In this case, insertion time is $O(n)$ since an interval can be inserted to all tree nodes in the worst case. On the other hands, a search can be conducted in $O(\log n)$ time. The other method is to decompose the interval into two parts, $x > a$ and $x < b$, then to insert each part into the tree separately. In this case, the predicate insertion can be done in an $O(\log n)$ time, while the search cost increases to $O(n)$ because all predicates in the tree can be included in the search results. Here, the former is used to implement the PB+-tree since we had more focus on insertion performance than search cost.
A data generator was also implemented to synthesize test data sets. The range of endpoint values of predicates was set to [0, 1000000] and up to 128,000 predicates were randomly produced with random inequality types (i.e. <, ≤, >, and ≥). Three types of data sets were used to simulate the worst, the general, and the best cases of the proposed method. The worst case set has only closed interval predicates while the best case set has only open interval predicates. The general case set has both open and closed interval predicates, and in this case, the ratio of the two kinds of predicates was set to 1:1. For each case, 10 sets of random intervals were generated and used to measure average insertion and search times of the three methods. The experiments were conducted on an Intel Pentium IV 2.4 GHz machine, running Window XP, with 1GB of main memory.

4.2 Results

First, the search performances of the PB+-tree, the IBS-tree and the proposed method are compared. The number of predicates in a set was increased from 8K to 128K, and average times of 100,000 searches over the predicates were measured for each method. Fig. 7 (a) shows their average search times obtained from the worst case data sets; the results from other types of data sets are omitted since they are similar to that provided. Note that the performance of the proposed method is similar to those of the PB+-tree and the IBS-tree although the proposed method performs more steps to obtain predicates from the slots \( \Lambda_i \) and \( \Gamma_i \). This leads to that the search performance is influenced by other factors such as examining retrieved predicates and copying them to a result list.

Then, the three methods were compared in terms of insertion performance. For the experiments, the number of predicates in a set was increased from 8K to 128K and the insertion times were observed. Figs. 7 (b)-(d) show the average insertion times when using the worst-case, the best-case and the general-case sets, respectively.

When only closed intervals were given (i.e. when using the worst-case sets), the IBS-tree and the proposed method showed the similar insertion performances while the insertion time of the PB+-tree increased steeply (Fig. 7 (b)); the insertion time of the PB+-tree when the number of predicates is larger than 16K is not depicted in the figure since the values are too high in those cases. On the other hands, when only open intervals were given, the PB+-tree and the proposed method showed the similar performances while the insertion time of the IBS-tree degraded significantly (Fig. 7 (c)). In this case, the proposed method showed on the average seven times better performance than the IBS-tree; for some cases, the performance gap of the two methods reached up to twenty times.

When both types of predicates were given (Fig. 7 (d)), the proposed method provided the best performance compared to other methods. When the ratio of open and closed intervals was the same as in Fig. 7 (d), the proposed method showed on the average 50% better performance than the IBS-tree. The performance gap increased as the ratio of open intervals increased. For example, when the ratio of two types of predicates was given to 2:1, the proposed method showed three times better performance than the IBS-tree. All of the above results can be explained by that the proposed method uses the shared lists for the insertion of open interval predicates while it utilizes an algorithm of the IBS-tree in the other case.
A HYBRID METHOD FOR FAST PREDICATE MATCHING

5. CONCLUSION

In this paper, a predicate indexing method that supports fast insertion of predicates was proposed based on the IBS-tree scheme without degrading the search performance. When processing the closed intervals, an algorithm of the IBS-tree whose insertion cost is $O(\log^2 n)$ was used. On the other hands, when handling the open intervals, the shared lists were used and the insertion time was reduced to $O(\log n)$ in this case. Consequently, the insertion performance of the proposed method gets better as the ratio of open intervals increases. Regarding this, two experimental results were provided which showed that (1) the proposed method provided approximately 50% better insertion performance than the IBS-tree when there were both open and closed interval predicates with the ratio of 1:1, and (2) the performance gap between the two methods was on the average seven times when there were only open interval predicates.

REFERENCES


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 received his Ph.D. degree in Computer Science from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea, in 2010. He is currently working at LG Electronics, Seoul, South Korea, as a senior research engineer. 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 Korea Advanced Institute of Science and Technology, 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.