DEWS2003 3-B-02

Design of Secondary Storage Predicate Index for Publish/Subscribe System

Botao WANG†, Wang ZHANG†, and Masaru KITSUREGAWA†

† Institute of Industrial Science, The University of Tokyo, Komaba 4–6–1, Meguro Ku, Tokyo, 135–8505 Japan

E-mail: †{botaow,zhangw,kitsure}@tkl.iis.u-tokyo.ac.jp

Abstract Efficient event matching algorithms are the core of publish/subscribe systems which require both fast search as well as efficient support for dynamic insertions and deletions. Such algorithms are typically designed based on memory structure for performance reasons. Given the explosive growth of information, it is not always practically feasible to keep the index for event filtering memory-resident, thereby necessitating the need for a secondary storage structure. To address this problem, we propose a B+tree-based predicate index for secondary storage structure with space complexity O(n) and time complexity O(log n) for search, insertion and deletion operations. The main idea is that grouping predicates by their operators (<, =, ! =, >) to share computation and for each pointer inside B+tree, a pointer flag that indicates the types(groups) of predicates which are kept in the leaf nodes, is added for search operation.

Key words Predicate Index, Event Matching, Publish/Subscribe System

1. Introduction

The rapid growth of technology has considerably changed the manner and scale of information management. Users can find and provide information easily through brokers like the Web; at the same time, in various applications, such as stock tickers, traffic control, network monitoring, Web logs and clickstreams, and Sensor networks, input data arrive as continuous ordered data streams [18]. There is a growing necessity for systems to be able to capture the dynamic aspect of information. Publish/subscribe systems provide subscribers with the ability to express their interests in an event in order to be notified afterwards of any event fired by a publisher, matching their registered interests. In other words, producers publish information on publish/subscribe systems and consumers subscribe to their desired information [8].

The precursor to publish/subscribe systems was subject-based. In such systems, information consumers subscribe to one or more subjects and the system notifies them whenever an event classified as belonging to one of their relevant subjects is published. A representative example of such a system is a mailing list. Thousands of mailing lists exist, encompassing a wide variety of topics. The user subscribes to lists of his/her interests and receives messages via mail. However, it offers only limited expressiveness. Given its coarse classification of topics, it provides *only* a crude granularity of interest matching, thereby resulting in the user receiving either too many irrelevant or too few relevant messages.

As an attractive alternative to subject-based publish/subscribe systems, content-based publish/subscribe systems typically introduce subscription schemes based on the properties of the given notifications. Notably, events are not classified according to some pre-defined subject name, but according to properties of the event themselves. For example, a content-based system for a stock market may define a subscription schema as a tuple containing three attributes: $CompanyName$, $Price$ and $ChangeRatio$ with string, float and float types respectively. The user can choose stock information by the values of attributes not by subject $Stock$ only.

The cost of the gain in expressiveness of content-based system is an increase in the complexity of the matching process. The efficiency of matching highly depends on matching algorithms. Input data arrive as data stream and the subscriptions are inserted and deleted dynamically.

As far as we know, in the context of publish/subscribe system, many matching algorithms are generally pro-

posed based on memory structure [1] [4] [9] [10] [14] [17] [19]. As pointed in [18], input data arrive in form of data stream, it's very difficult to keep all the index data in the memory practically. At the same time, for the similarity of operations(range or interval query), some searching algorithms designed for active database [12], spatio-temporal database [3] [6] [7] [11] can be applied to publish/subscribe system, those algorithms are not designed originally for publish/subscribe system. They lack flexibility of insertion and deletion and don't support relational operator "!=" directly.

In this paper, we proposed a secondary storage predicate index structure based on B+tree for efficient event matching. The rest of this paper is organized as follows. Section 2 formally defines the event matching problem. Section 3 introduces the related work. Section 4 describes our proposed predicates index structure: PB+tree and event matching algorithm in this context. In Section 5, analytical comparisons between our proposed algorithm and existing techniques are made. Finally, conclusion is given in Section 6.

2. Event Matching Model

The event matching problem can be expressed as follows. Given an event e and a set of subscriptions S, determine all subscriptions in S that are matched by e. A subscription is a conjunction of predicates. A predicate is a triple consisting of an attribute, a constant, and a relational operator (<, <=, =, !=, >=, >). A subscription schema defines the type of the information to be supported by publish/subscribe system. The attributes are defined in subscription schema. For example, three attributes: $CompanyName$, $Price$ and $ChangeRatio$ with string, float and float types respectively can be defined for stock market. Following is a subscription example of stock schema, ($CompanyName = Yahoo$) AND ($Price$>1000) AND ($ChangeRatio$<0.05). An event is an array of pairs of (Attribute, Constant). The size of array depends on subscription schema. Following is an event example of stock schema, ($CompanyName$, $Intel$), ($Price$, 5000), ($ChangeRatio$, 0.03). An event e matches a subscription S if all predicates in S are satisfied by some (Attribute, Value) pairs in e. For example, event ($CompanyName$, $Yahoo$), ($price$, 500), ($ChangeRatio$, 0.1) matches following subscription which is expressed as a conjunction of two predicates: ($CompanyName = Yahoo$) AND ($Price < 1000$).

Event matching algorithms in content-based publish/subscribe systems can be classified into two categories:

- Algorithms based on predicate index. The algorithms based on predicate indexing consist of two steps:
  - The first step determines all predicates that are satisfied by the event.
  - The second step finds all subscriptions that are matched by the events based on the results of the first phase.

  Algorithms based on predicate indexing techniques use a set of one-dimensional index structure to index predicate in the subscriptions. They differ from each other by the way to select predicates from subscriptions, which are kept in the index structures [4] [9] [10] [12] [14] [17] [19].

  Basically, the predicates are grouped based on all subscriptions. A predicate family consists of predicates having the same attribute. For each attribute, one predicate index is built. For example, for stock schema introduced previously, three predicate indexes will be built for $CompanyName$, $Price$, $ChangeRatio$.
- Algorithm based on subscription index [1] [15]. The techniques based on subscription index insert subscriptions into a matching tree. Events enter the tree from root node and are filtered through by intermediate nodes. An event that passes all intermediate testing nodes reaches leaf nodes where references of matching subscriptions are stored.

Our proposed data structure is designed for predicate index. Although, there are many proposals for selection of predicates from subscriptions. [9] [10] [12] [19], the predicate index is essential while determines all the predicates that are satisfied by the event at the first step. From next introduction, we will concentrate on the predicate index for one predicate family without considering about the second step. For details of different methods of predicates selection, please refer to [9] [10] [12] [19].

3. Related Work

A lot of algorithms related to event matching have been proposed. Some are proposed for publish/subscribe systems [1] [9] [10] [16] [15] [19] and continuous queries [4] [5] [18]; Some are proposed for active database [12] [13] [14], spatio-temporal database [3] [7] [11].

In [9] [10] [19], predicate indexs are built. The algorithm consist of two phases: the first step gets satisfied predicates, the second step collects matching subscriptions according to the results of the first step. In [9], three predicate indexes are built for operators (=, >, <). For = operator, hash table or binary search can be used. For > and < operator, binary search trees are used. In [9], hash table is used to build index for predicates with = operator. [19] is a information Dissemination System(IDS) for

document filtering. There, predicate index is a inverted list which is built based on the vocabularies used in predicates.

Different from predicate index, [1] and [15] built subscription tree based on subscription schema. In [1], each non-leaf node contains a test, and edges from the node represent results of that test. The test and result corresponds to predicate. A leaf node contains a subscription. The matching is to walk the matching tree by performing the test prescribed by each node and following the edge according to the result of test. if number of matched subscription is greater then one, multiple paths will be walked. In [15], Profile(subscription) tree is built, the height of tree is number of attributes defined in subscription schema. Each non-leaf level corresponds to one attribute of event schema. Each attribute domain is divided non-overlapping subrange by the value of predicate. One leaf node contains multiple subscriptions whose predicates are satisfied by the values of attributes in the subranges. There is only a single path to follow in order to find the matched subscriptions.

In [12] [13] [14], algorithms related to rule management were proposed. The key component of the algorithm in [12] is the interval binary search tree(IBS-tree). The IBS-tree is designed for efficient retrieval of all intervals that overlap a point, while allowing dynamic insertion and deletion of intervals. In [13], the same idea of IBS-tree is implemented by skip lists. "Expression Signature" is designed to group subscriptions and share computation in [14].

Event filtering is critical step of continuous queries. In [5], Expression Signature is used to group queries for computation sharing. In [17], four data structures: a greater-than balanced binary tree, a less-than balanced binary tree, an equality hash-table, and an inequality hash-table were built(we call them data structure Group Filter in Section 5). The structures are similar to that of [9], but inequality operator(!=) is supported. In [4], predicate index is built based on Red-Black tree. Each node contains five arrays that store queryIDs of the corresponding predicates. Five relational operators (<, <=, =, >=, >) are supported directly.

Because the range query of spatio-temporal database uses operators (<, <=, >=, >) in the similar way of predicate index, the related data structures can be used to built predicate index. In [7], an index structure for time interval is built. A set of linearly ordered indexing points is maintained by a B+tree, and for each point, a bucket of pointers refers to the associated set of intervals. In [3], Interval B+tree is built and the lower bounds of the intervals are used as primary keys. Multi-dimensional Rtree [11] and its variants may not behave well for one-dimensional interval for the reason of overlap of interval.

4. PB+tree: Predicate Index Based On B+tree

In this section, we will introduce our predicate index based on B+tree. We call it PB+tree.

4.1 Motivation

B+tree is a popular index structure for secondary storage. Besides its simplicity of implementation and maintenance, all the data are kept in the leaf nodes which are linked in an order list. Our idea is that build a predicate index for secondary storage based on B+tree and make use of the order of leaf node list to share computation. At the same time, all the predicates of one predicate family are kept in one data structure and support relational operators (<, <=, =, >=, >, !=) directly.

4.2 Structure of Predicate Index

The basic structure of predicate index is shown in Fig.1. The constant defined in predicate is used as key of B+tree. As shown in Fig. 1, data structure of B+tree is extended and three lists are added below leaf node list of B+tree.

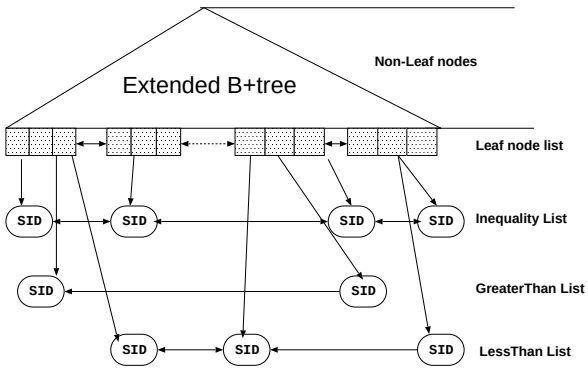

Figure 1 Basic Structure of Predicate index

Pointer inside Extended B+tree As shown in Fig. 2, pointer flag is added to every pointer pointing to child node in original B+tree. Pointer flag is a 3 bits array and each bit corresponds to one kind of predicates. If bit value is 1, it means that the predicates with corresponding type(GreaterThan, LessThan, Inequality) are kept in the leaf node of its subtree. The flag is used to guide search operation.

Nonleaf Node of Extended B+tree Fig.3 shows the nonleaf node structure of extended B+tree. A head is added to original B+tree's nonleaf node.

GreaterThan Counter records number of pointers whose GreaterThan bits are 1. It's used to fast operation of insertion and deletion.

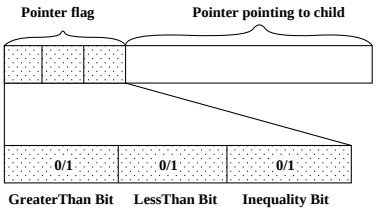

Figure 2 Pointer Structure Inside Extended B+tree

First GreaterThan Pointer is the first pointer whose GreaterThan bit is 1 in the node. The key related to this pointer is the smallest among the keys related to the pointers with GreaterThan Bits value 1. Last GreaterThan Pointer is the last pointer whose GreaterThan bit is 1 in the node. The key related to this pointer is the largest among the keys related to pointers with GreaterThan Bits value 1. They are used to fast search introduced in Fig. 13

The definitions of LessThan and Inequality parts in the head are similar to those of GreaterThan.

Notice that, there is only one inequality related field: Inequality Counter, because Inequality list doesn't require order.

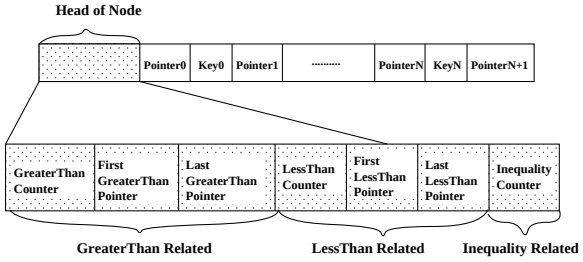

Figure 3 Node Structure of Extended B+tree

Leaf Node of Extended B+tree As shown in Fig.1, three lists are added below leaf node list of B+tree. The nodes on the lists are pointed by the pointers in leaf nodes. The data structure of original B+tree leaf node is extended as shown in Fig.4

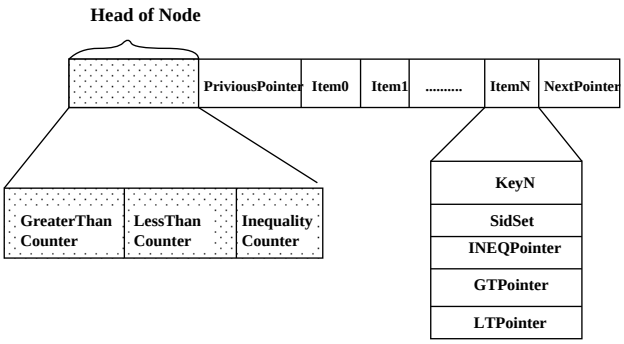

Figure 4 Data Structure of Leaf Node of

The definitions of PreviousPointer, Key and nextPointer are same as those in B+tree respectively. The SidSet is set of subscriptions which contain predicate in the form $Attribute = Key$, $Attribute <= Key$ or $Attribute >= Key$. INEQpointer, GTPointer. LTPointer are the pointers of Inequality list, GreaterThan list and LessThan list respectively. They are entry points to get results of predicate search according to input key. The value of each above pointer will be set *Null* if the predicate with corresponding operator and key doesn't exist. Notice that, besides equality operator (=), the Ids of subscriptions with predicate containing operators(<=, >=), are kept inside item of leaf node too.

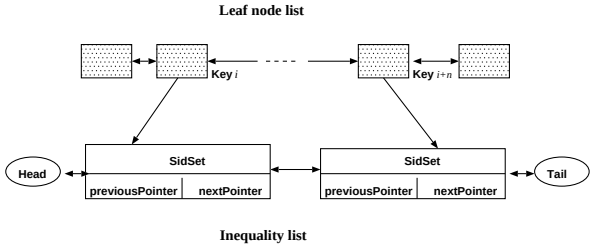

Figure 5 Data Structure of Node of Inequality List

Inequality List Inequality list is a list to deal with predicates with operators (!=). The data structure of its node is shown in Fig.5. It is used to match predicate in the form $Attribute! = Key$. SidSet is ID set of subscriptions which contain predicate $Attribute! = Key$. Double links are defined for search, insertion and deletion. Head pointer and Tail pointer are predefined before the index is created. previousPointer and nextPointer point to nearby node in two directions.

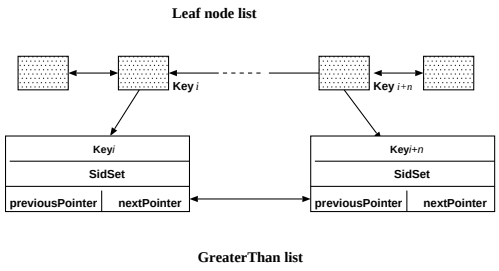

Figure 6 Data Structure of Nodes of GreaterThan List

GreaterThan List GreaterThan list is an order list to deal with predicates with operators ($>, >=$). The data structure is same as that of Inequality List. Here, SidSet is ID set of subscriptions which contain predicate $Attribute > Key$ or $Attribute >= Key$. For two GreaterThan predicates with different constants, the range represented by one predicate will totally cover another. It is determined by the value of the constant. For example, $Attribute > 10$ is true means $Attribute > 5$ is

true too. This property is used to share computation by get results from the point with the biggest cover. Same as original B+tree, the keys in one node and the nodes in the list are arranged in ascending order of the key. Previous-Pointer is the pointer pointing to the GreatThan list node with keys less than those of current node. NextPointer points to GreaterThan list node with keys greater than those of current node.

LessThan List LessThan list is an order list to deal with predicates with operators (<, <=). It has similar data structure with GreaterThan list.

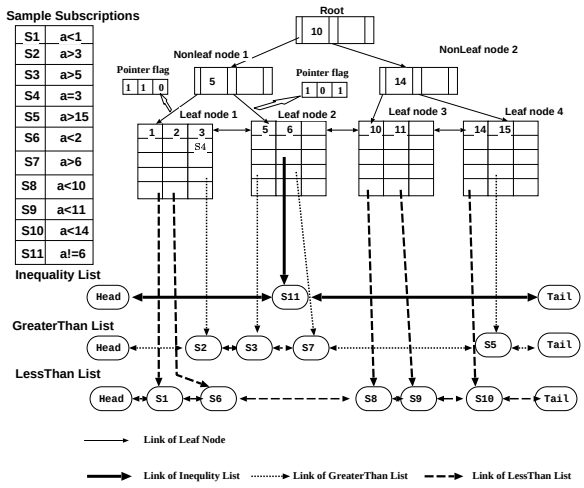

Figure 7 Example of Predicate Index

4.3 Searching Algorithm

An example of predicate index tree is shown in Fig.7. In order to be easy to understand, there is no duplication of predicates in the example.

Because there are mainly four kinds of predicates (Equality, Inequality, GreaterThan, LessThan), there are maximum four search paths(Equality path, Inequality path, GreaterThan path, LessThan Path) corresponding to four kinds of predicates. The four search paths start at root node and end at same/different leaf nodes. The leaf nodes on the search paths are called **EntryNode**. The item in **NntryNode** which is used as starting point to collect results is called **EntryItem**. In the case that four search paths end at more than one leaf node, the node where search paths split, is called **BranchNode**. Naturally, Equality path is same as search path of original B+tree. The other three paths depend the results of Equality path. For each search path, there exists one pointer pointing to its BranchNode. If there is not search path splitting (same as Equality path), the pointer of corresponding BranchNode is Null.

According to the sample shown in Fig.7, given $InputKey$ is 5,

- **Equality Path** The Equality path is { Root, Nonleaf node 1, Leaf node 2 }, which is same as search path of original B+tree. Its EntryItem with key 5 on Leaf node 2.
- **Inequality Path** The Inequality path is {Root, Nonleaf node 1, Leaf node 2}. The results on Inequality list is total Inequality list, because the INEQPointer of item with key 5 is Null. The pointer of its BranchNode is Null. Its EntryItem with key 5 on Leaf node 2.
- **GreaterThan Path** The GreaterThan path is {Root, Nonleaf node 1, Leaf node 2}. The results on GreaterThan list are nodes between Head and S3. The pointer of its BranchNode is Null. Its EntryItem is the one with key 5 on Leaf node 2.
- **LessThan Path** The LessThan path is {Root, Nonleaf node 1, Leaf node 1}. As shown in Fig.7, the LessThan Bit of the pointer pointing to Leaf node 2 is 0. So from Nonleaf node 1, it can be found that there is no LessThan predicate on Leaf node 2, the search paths should split here. The results on LessThan list are nodes between S6 and Tail. Its EntryItem is the one with key 2 on Leaf node 1. The pointer of its BranchNode is Nonleaf node 1.

In above example, Nonleaf node 1 is BranchNode because search paths split there; Leaf node 1 and Leaf node 2 are EntryNodes.

The main search algorithm of PB+tree is shown in Fig. 8.

```
Search(InputKey, Root,Result)
1  //Input:
2  //        InputKey: input data from event,
3  //        Root: root of B+tree
4  //Output:
5  //        Result: Pointer of Set of Sid
6   Reset  Result,  BranchNodeOf InequalityPath
7          BranchNodeOfGreaterThanPath And  BranchNodeOfLessThanPath
8   // set pointer of above variables to be Null
9  EntryNode = GetEqualityPredicates(InputKey, Root,Result)
10  //Get predicates realted to equality operation
11  GetInequalityPredicates(BranchNodeOfInequalityPath,
12                          EntryNode, InputKey, Result)
13  //Get predicates related to inequality
14  GetGreaterThanPredicates(BranchNodeOfGreaterThanPath,
15                           EntryNode, InputKey, Result)
16  //Get predicates related to greater than operation
17  GetLessThanPredicates(BranchNodeOfLessThanPath,
18                        EntryNode, InputKey, Result)
19  //Get predicates related to less than operation
```

Figure 8 Main Search Algorithm

Notice that in the case that search paths split, the pointers of variables
$BranchNodeOfInequalityPath$,
$BranchNodeOfGreaterThanPath$,
$BranchNodeOfLessThanPath$

will be set inside function **GetEqualityPredicates** shown in Fig.9.

GetEquality Predicates As mentioned in section 4.3, equality path is same as that of original B+tree. Line 9 uses original algorithm of B+tree to find pointer for next level. If expressions at line 10, 13 or 16 are true, it means the search paths split here. So the pointer of corresponding BranchNode should be kept for search of other types of predicates. So deos the returned EntryNode of Equality Path at line 24.

```
GetEqualityPredicates( InputKey, Root, Result)
1  // Input:
2  //      InputKey: input data from event
3  //      Root: root of B+tree
4  //Output:
5  //      Result: Set of Sid
6  //      EntryNode: entry node of  equality path
7   Assign  Root to CurrentPointer
8   While CurrentPointer points  nonleaf node
9       Get ChildPointer in  the original way of   B+tree search algorithm
10     If GreaterThan Bit of  ChildPointer and CurrentPointer is different
11          Assign CurrentPointer to BranchNodeofGreaterThan
12     Endif
13     If  LessThan Bit of  ChildPointer and CurrentPointer is different
14          Assign CurrentPointer to BranchNodeofLessThan
15     Endif
16     If  Inequality Bit of  ChildPointer and CurrentPointer is different
17          Assign CurrentPointer to BranchNodeofInequality
18     Endif
19     Assign ChildPointer to CurrentPointer
20  EndLoop
21  If  InputKey is found in the leaf node pointed by CurrentPointer
22      Add content of  SidSet in  item with key value InputKey  into Result
23 Endif
24 Return CurrentPointer //return entry node of Equality Path
```

Figure 9 Algorithm of Function GetEqualityPredicates

GetInequality Predicates As shown in Fig.10, this algorithm is relatively simple. If pointer of Branchnode of Inequality is Null, it means it has possibility to get EntryItem of Inequality list. If EntryItem is found, then collect results from INEQPointer of EntryItem. In other cases, all the nodes on Inequality list belong to result of this search.

GetGreaterThanPredicates There are two cases in Function shown in Fig.11.

- Case 1: It has same search path as Equality Path.

In this case, pointer of BranchNode of GreaterThan Path is Null. $GTEntryItem$ is EntryItem to collect results on GreaterThan list and it is gotten at line 11 by function **GetGTEntryItem**. The results are collected at lines 18-19.

- Case 2: It has different search path from Equality Path.

In this case, the EntryNode of should be gotten first and it is gotten by function **GetGTEntryNode** at lines14-15. The others are same as case 1.

```
GetInequalityThanPredicates(BranchNodeOfInequalityPath,
                            EntryNode, InputKey, Result)
1  // Input:
2  //      BranchNodeOfInequalityPath: Pointer of branch node
3  //                                  of  InequalityPath
4  //      EntryNode: EntryNode of Equality Path
5  //      InputKey: input data from event
6  // Output:
7  //      Result: Set of Sid.
8  //
9  If  (BranchNodeOfInequalityPath is Null
10                and  InputKey is found in EntryNode)
11     Add all nodes on Inequality list except the node
12           corresponding to key InputKey into Result
13 Else
14     Add all nodes on Inequality list into Result
15 Endif
```

Figure 10 Algorithm of Function GetIequalityPredicates

```
GetGreaterThanPredicates(BranchNodeOfGreaterThanPath,
                         EntryNode, InputKey, Result)
1   //Input:
2   //      BranchNodeOfGreaterThanPath: Pointer of Branch node of
3   //                                   GreaterThan Path
4   //      EntryNode: Pointer of entry node of equality path
5   //      InputKey: input data from event
6   // Output
7   //      Result:set of Sid
8   //
9   If BranchNodeOfGreaterThanPath is Null
10     //Operation on leaf  node
11     GTEntryItem = GetGTEntryItem(InputKey, EntryNode)
12  Else
13     // Operation on nonleaf  node
14     GTEntryNode = GetGTEntryNode(InputKey,
15                                  BranchNodeOfGreaterThanPath)
16     GTEntryItem = GetGTEntryItem(InputKey,  GTEntryNode)
17  Endif
18  Put nodes between head  node and node pointed by  GTPointer
19     of the GTEntryItem into Result.
```

Figure 11 Algorithm of Function GetGreaterThanPredicates

```
GetGTEntryItem(InputKey, EntryNode)
1  //Input:
2  //   InputKey:input data from event
3  //   EntryNode:  Entry Node of  GreaterThan search path
4  //Output:
5  //   EntryItem: Entry Item
6  Assign the item which contains the largest key among
7        the items whose GTPointers are not empty to LargestItem.
8  If  (InputKey  is greater than the Key of LargestItem)
9        Return LargestItem.
10 Else
11       Search for GTEntryItem from LargetItem in descending
12            order on EntryNode. The  GTEntryItem is the  last met
13            item whose key is greater than or equal to InputKey and
14            has no empty  GTPointer.
15       Return GTEntryItem
16 Endif
```

Figure 12 Algorithm of Function GetGTEntryItem

In order to understand algorithm in Fig.12, please refer to leaf node 2 in Fig.7, there $LargestItem$ is item with 6. If $InputKey$ is 7, then $GTEntryItem$ is the item with key 6 which is gotten by line 8-9. If $InputKey$ is 5, then $GTEntryItem$ is the item with key 5 which is gotten by line11-15.

```
GetGTEntryNode(InputKey, BranchNodeOfGreaterThanPath)
1  //Input:
2  //      InputKey: input data from event
3  //      BranchNodeOfGreaterThanPath: pointer of branch node of
4  //                                   GreaterThan Path.
5  //Output:
6  //      GTEntryNode: Entry Node of GreaterThan Path
7
8   Assign BranchNodeOfGreaterThanPath to CurrentNode
11  While CurrentNode is not leaf node
12      In CurrentNode, check all pointers whose GreaterThan Bit is 1,
              get ChildNode on GreaterThan Path in the same way
              as original B+tree algorithm.
13      Assign ChildNode to CurrentNode
14  EndLoop
33  Return CurrentNode
```

Figure 13 Algorithm of Function GetGTEntryNode

Fig.13 shows the function to get EntryNode of GreaterThan path. The algorithm is similar to search algorithm of original B+tree. Because it is traverse of GreaterThan path, so the difference is that only pointers with GreaterThan Bit 1 will be used as candidates to be checked to get child node.

GetLessThanPredicates Because algorithm of function **GetLessThanPredicates** has same idea as function**GetGreaterThanPredicates**, the details of the algorithm is skipped.

4.4 Insert Algorithm

Insert algorithm has two main steps:

- Step1: insert key to leaf node list. This procedure is similar to that of original B+tree insertion. The extra work is maintenance of pointer flag and information in head of nodes.
- Step2: According to the operator of predicate, insert ID of subscription into corresponding list. Because the GreaterThan list and LessThan list are order lists, so the algorithm to find EntryItem can be utilized to find proper insertion point where the new node is inserted according to ascending order of the key. The Inequality list doesn't require order, so the new node may be inserted at *Head* node or *Tail* node of Inequality list.

4.5 Delete Algorithm

The delete algorithm is a reverse procedure of insertion. It's similar to deletion of original B+tree. The details are skipped here.

5. Analytical Comparison

Assume the number of unique predicate is n and the total number of predicates satisfied by event is L. As introduced in Section 4.3, there are maximum 4 search paths in order to get EntryItem, the search time complexity to get EntryItem is $O(\log n)$. So the total search time complexity is $O(\log n + L)$.

As introduced in Section 4.4, there are two stepsin insertion operation, both steps have same time complexity $O(\log n)$, so the time complexity of insertion is $O(\log n)$. For the same reason, the complexity of deletion is $O(\log n)$ also.

Table 1 Comparisons of Space and Time Complexities

Algorithm	Space	Search	Insert/Delete
PB+tree	$O(n)$	$O(\log n + L)$	$O(\log n)$
Grouped Filter [17]	$O(n)$	$O(logn + L)$	$O(\log n)$
Red-Black tree based [4]	$O(n)$	$O(n)$	$O(\log n)$
IBS-Tree [12]	$O(n \log n)$	$O(\log n + L)$	$O(log^2 n)$
Time Index [7]	$O(n^2)$	$O(\log n + L)$	MIN:$O(\log n)$ MAX:$O(n)$
Interval B+tree [3]	$O(n)$	$O(n)$	$O(\log n)$

Our algorithm is built based on B+tree, the leaf nodes are extended to link three lists, so the space complexity is $O(n)$. In table 1, the comparisons of complexities are listed.

Considering Complexities of time and space, we compare with only Grouped Filter [4] for its best complexities. As introduced in Section 3, Grouped Filter uses four data structures for different operators and PB+tree has only one data structures. Grouped Filter is a main memory predicate index and PB+tree is designed for secondary storage predicate index.

Besides differences in space and time complexities, the data structure designed for spatio-temporal database is used to find all intervals that intersect a input point, which means they support predicates with format ($Constant_{Start} < Attribute < Constant_{End}$). In the case that $Constant_{Start}$ or $Constant_{End}$ is infinite, overlap will rise greatly for IBS-tree [12] and Time Index [3], search efficiency will decline greatly for Interval B+tree. They don't support predicate with single operator directly.

By the comparisons in Table 1 and above analysises, we can conclude that efficient event matching can be reached by building secondary storage predicate index on PB+tree.

6. Conclusion

In this paper, we introduced a secondary storage predicate index based on B+tree and algorithms for event matching. The index supports predicates with relational operators($<, <=, =, !=, >=, >$). The space complexity is O(n). The time complexity of search is $O(\log n + L)$ and the time complexity of both insertion and deletion is $O(\log n)$. Analytical comparison of our proposed algorithms with existing work indicates that our secondary storage predicate index is efficient for event matching.

References

[1] Marcos K.Aguilera, Robert E.Strom, Daniel C. Sturman, Mark Astley, Tushar D.Chandra. Matching Events in a Content-based Subscription System. Eighteenth ACM Symposium on Principles of Distributed Computing(PODC), 1999

[2] M.deBerg, M.van Kreveld, M.Overmars, O.Schwarzkopf. "Computational Geometry-Algorithms and Applications". ISBN 3-540-65620-0 Springer. 1998

[3] Tolga Bozkaya, Meral Ozsoyoglu. Indexing transaction time database. Information Sciences 112(1998)

[4] Sirish Chandrasekaran, Michael J. Franklin. Streaming Queries over Streaming Data. Proceedings of the 28th VLDB Conference, Hong Kong, 2002

[5] Jiangjun Chen, David J. DeWitt, Feng Tian, Yuan Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. ACM SIGMOD 2000

[6] Y.-J. Chiang and R.Tamassai, "Dynamic Algorithms in Computational Geometry". Technial Report CS-91-24, Dept. of Computer Science, Brown Univ., 1991

[7] Ramez Elmasri, Gene T.J. Wuu, Yeong-Joon Kim. THE TIME INDEX: AN ACCESS STRUCTURE FOR TEMPORAL DATA. VLDB 1990

[8] P. Th. Eugster, P. Felber, R. Guerraoui and A.-M. Kermarrec. The Many Faces of Publish/Subscribe. Technical Report 200104, Swiss Federal Institute of Technology

[9] Francoise Fabret, Francois Llirbat, Joao Pereira, Dennis Shasha. Efficient matching for Content-based Publish/Subscribe Systems. Technical report, INRIA, 2000.

[10] Francoise Fabret, H.Arno Jacobsen, Francois Llirbat, Joao Pereira, Kenneth A.Ross, Dennis Shasha. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems. ACM SIGMOD 2001

[11] Antonin Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. ACM SIGMOD 1984

[12] Eric N. Hanson, Moez Chaaboun, Chang-Ho, Yu-Wang Wang. A Predicate Matching Algorithm for Database Rule Systems. ACM SIGMOD 1990

[13] Eric N. Hanson, Theodore Hohnson. Selection Predicate Indexing for Active Database Using Interval Skip List. TR94-017. CIS department, Univeristy of Florida, 1994

[14] Eric N. Hanson, Chris Carnes, Lan Huang, Mohan Konyala, Lloyd Noronha. Scalable Trigger Processing. ACM SIGMOD 1999

[15] Annika Hinze, Sven Bittner. Efficient Distribution-Based Event Filtering. International Workshop on Distributed Event Based Systems. Austrai July 2002

[16] H.Arno Jacobsen, Francoise Fabret. Publish and Subscribe Systems. Tutorial. ICDE 2001

[17] Samuel Madden, Mehul Shah, Joseph Hellerstein, Vijayshankar Raman. Continuously Adaptive Continuous Queries(CACA) over Streams. ACM SIGMOD 2002

[18] Rajeev Motwani. Models and Issues in Data Stream Systems. Invited Talk. PODS 2002

[19] Tak W.Yan, Hector Garcia-Molina. The SIFT Information Dissemination System. In ACM TODS 2000