Design of B+tree-Based Predicate Index for Efficient Event Matching

Botao Wang[1], Wang Zhang[1], and Masaru Kitsuregawa[1]

Institute of Industrial Science, The University of Tokyo
Komaba 4–6–1, Meguro Ku, Tokyo, 135–8505 Japan
{botaow, zhangw,kitsure}@tkl.iis.u-tokyo.ac.jp

Abstract. Efficient event matching algorithms are the core of publish/subscribe systems. 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. Incidentally, even though search algorithms designed for active databases and spatio-temporal databases are applicable to publish/subscribe systems, these algorithms are *not* specifically designed for publish/subscribe systems which require both fast search as well as efficient support for dynamic insertions and deletions. To address this problem, we propose a predicate index for secondary storage structures with space complexity $O(n)$ and search time complexity $O(\log n)$. Analytical comparison of our proposed algorithms with existing work indicates that our secondary storage predicate index is efficient for event matching.

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[17]. There is a growing necessity for systems to be able to capture the dynamic aspect of Web information. Publish/subscribe systems provide subscribers with the ability to express their interest in an event in order to be notified afterwards of any event fired by a publisher, matching their registered interest. In other words, producers publish information on publish/subscribe systems and consumers subscribe to their desired information[7].

The precursor of 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.

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 names, 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, many matching algorithms in the context of publish/subscribe are generally proposed based on memory structure [1][3][8][9][13][16][18]. As pointed in [17], 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[11], spatio-temporal database [2][5][6][10] can be applied to publish/subscribe system, those algorithms are not designed originally for publish/subscription system and lack of flexibility of insertion and deletion, don't support relational operator "!=" directly.

In this paper, we will propose 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 predicate 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 properties of the information to be supported by publish/subscribe system. 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, ($CompanyName$ = $Yahoo$) AND ($Price$>1000) AND ($ChangeRatio$<0.05). An event is an array of (Attribute, Constant) tuples. The size of array depends on the number of attributes defined in 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) tuples in e. For example, the event $(CompanyName, Yahoo)$, $(price, 500)$, $(ChangeRatio, 0.1)$ matches the 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 predicate index. The solutions based on predicate indexing consist of two phases:
    - The first phase determines all predicates that are satisfied by event.
    - The second phase finds all the subscriptions matched by the event according to the results of the first phase.

  Algorithms based on predicate indexing techniques use a set of one-dimensional index structures to build indexes for predicates defined in subscriptions. They differ from each other in the way of selecting predicates from subscriptions to index structures[3] [8][9][11][13] [16][18].

  Basically, predicates are grouped by attributes. A predicate family consists of predicates with same attribute. For each attribute, one predicate index is built. For example, for stock schema introduced previously, three predicate indexes will be built on attributes $CompanyName$, $Price$, $ChangeRatio$.
- Algorithms based on subscription index[1][14]. The techniques based on subscription index insert subscriptions into a decision tree. Events enter the tree from root node and are filtered through by intermediate nodes. An event that passes all intermediate testing nodes reaches a leaf node where reference(s) of matched subscriptions are stored.

Our index data structure is designed for predicate index. Although, there are many proposals for predicates selection [8][9][11][18], the predicate index is essential while getting all satisfied predicates according to the event at the first phase. In the following introductions, we will concentrate on the predicate index of one attribute without considering about the second phase. For details of different selection methods of predicates, please refer to [8][9][11][18].

3 Related Work

A lot of algorithms related to event matching have been proposed. Some are proposed for publish/subscribe systems[1] [8] [9][15] [14][18] and continuous queries[3] [4][17]; Some are proposed for active database [11] [12][13], spatio-temporal database [2] [6] [10].

In [8][9][18], 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 [8], 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 [8], hash table is used to build index for predicates with = operator. [18] 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 [14] 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 [14], 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 [11][12][13], algorithms related to rule management were proposed. The key component of the algorithm in [11] 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 [12], the same idea of IBS-tree is implemented by skip lists. "Expression Signature" is designed to group subscriptions and share computation in [13].

Event filtering is critical step of continuous queries. In [4], Expression Signature is used to group queries for computation sharing. In [16], 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 [8], but inequality operator(!=) is supported. In [3], 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[6], 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 [2], Interval B+tree is built and the lower bounds of the intervals are used as primary keys. Multi-dimensional Rtree[10] 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

4.1 Motivation

B+tree is an efficient secondary storage index structure and all its data are kept in leaf nodes which are linked in an order list. Our idea is that to build a predicate index on secondary storage based on B+tree and make use of the order of leaf node list to share computation. All the predicates of one attribute are kept in one extended B+tree which supports relational operators (<, <=, =, >=, >, !=) directly.

4.2 Structure of Predicate Index

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

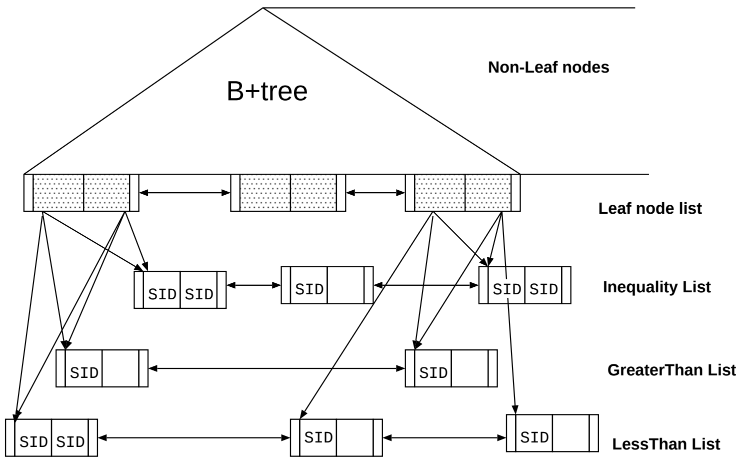

Fig. 1. Basic Structure of PB+tree

Inequality List Inequality list is used to deal with predicates with operator (!=). The data structure of its node is shown in Fig.2. Mainly, it consists of two pointers and an array of items. An item consists of key and SidSet. The key is the same as the key used in B+tree. It corresponds to predicate $Attribute! = Key$. SidSet is ID set of subscriptions which contain predicate $Attribute! = Key$. Because $Attribute! = Key$ means a special range ($Attribute > key$ AND $Attribute < Key$) double links are defined in inequality list. Inside one node, the items are arranged in descending order of key. PreviousPointer points to previous node of inequality list in ascending order of key. NextPointer points to next node of inequality list in descending order of key. Logically, Inequality list is an order list of items, where the logically adjacent two items maybe be kept in two different adjacent Inequality list nodes.

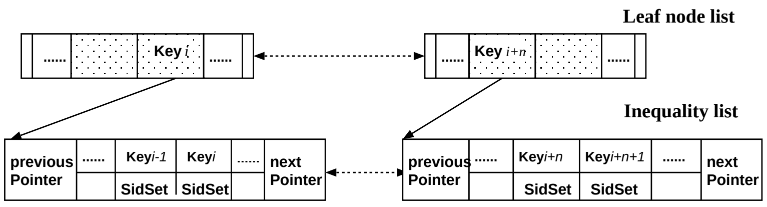

Fig. 2. Data Structure of Inequality List Node

GreaterThan List and LessThan List GreaterThan list is used to deal with predicates with operators ($>, >=$). The data structure of GreaterThan list node is shown in Fig.3 It is similar to that of Inequality list. The difference is that it has only previousPointer, no nextPointer. For each item, SidSet is ID set of subscriptions which contain predicate $Attribute > Key$ or $Attribute >= Key$. For two predicates with different keys, the range represented by one predicate will totally cover another on GreaterThan list. It is determined by the value of key. For example, $Attribute > 10$ is true means $Attribute > 5$ is true too. We make use of this property to share computation to collect results on GreaterThan list via previousPointer from the item corresponding to the largest range designated by input.

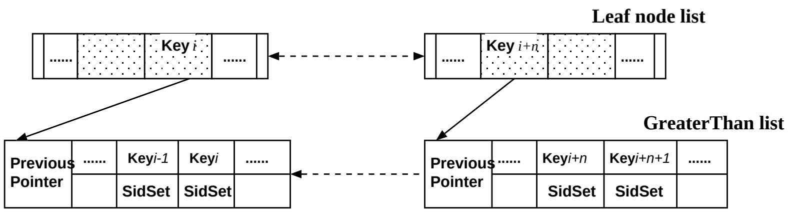

Fig. 3. Data Structure of GreaterThan List Node

LessThan List has similar data structure to GreaterThan List. The difference is that the pointer of node is nextPointer, not previousPointer.

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

It consists of an array of items, previousPointer and nextPointer. The definitions of previousPointer, key and nextPointer are same as those in B+tree respectively. For each item, besides the key, the SidSet is ID set of subscriptions which contain predicates in the forms of $Attribute = Key$, $Attribute <= Key$ or $Attribute >= Key$. INEQPointer, GTPointer, LTPointer are the pointers of Inequality list node, GreaterThan list node and LessThan list

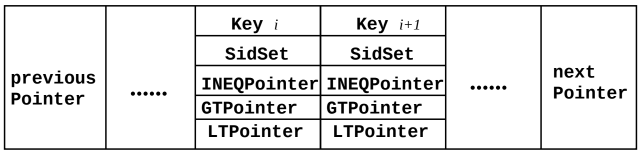

Fig. 4. Data Structure of Leaf Node

node respectively. GTPointer points to the GreaterThan list node where exits the item with the biggest key less than or equal to the key kept in the item of the leaf node. For example, GTPointers of key 5 and 6 in Fig. 5. LTPointer points to the LessThan list node where exists the item with the smallest key greater than or equal to the key kept in the item of the leaf node. Because Inequality predicate represent two special ranges and has double links, INEQPointer can be set by the same way as that of GTPointer or LTPointer to keep order. In Fig.5, the setting of INEQPointer is the same as that of GTPointer.

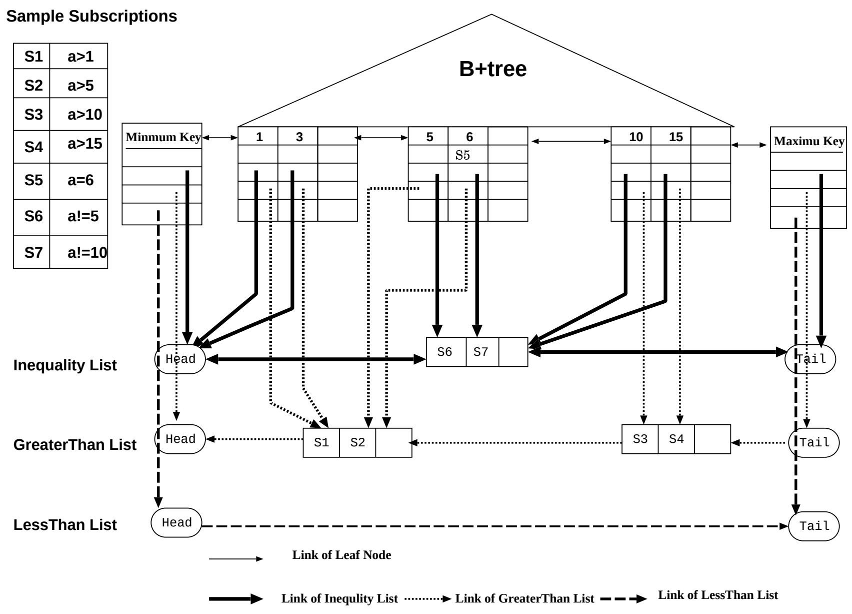

Fig. 5. Example of B+tree-Based Predicate Index

An example of predicate index tree is shown in Fig.5. On each list, there are both head and tail nodes exist where the keys are Minimum value and Maximum value in head and tail node respectively.

4.3 Insert Algorithm

```
Insert(Sid, Predicate, Root)
1  //Sid:Identifier of Subscription. Predicate:input predicate.
2  //Root: root of B+tree
3  Insert Predicate.constant into B+tree
4  Assign  the leaf node holding Predicate.constant to CurrrentLeafNode
5  IF (Predicate.constant didn't exist before insertion)
6     // initialize pointers of new inserted item on the leaf node
7     Assign the item with key Predicate.constant
8                                            to CurrentItem
9     Assign the item with biggest key less than
10                                  Predicate.constant to PreviousItem
11    Assign the item with smallest key  greater than
12                                  Predicate.constant to NextItem
13    //Set pointers according to nearby item.
14    Assign GTPointer of PreviousItem to GTPointer of CurrentItem
15    Assign LTPointer of NextItem to LTPointer of CurrentItem
16    Assign INEQPointer of PreviousItem to INEQPointer of CurrentItem
17  ENDIF
18  InsertToList(SID, Predicate, CurrentLeafNode)
```

Fig. 6. Algorithm of Inserting Leaf Node List

Insert Algorithm has two main steps, 1)the first is insertion of leaf node list of B+tree, where INEQPointer, GTPointer or LTPointer of the new item should be set. 2)the second is insertion of one of the three lists according to the operator of predicate. Besides inserting *Sid* of new subscription to SidSet. INEQPointers, GTPointers or LTPointers of related items on leaf nodes should be adjusted as shown in Fig.7(Line 7) according to input.

The first step is shown in Fig. 6. Line 3 inserts *Predicate.constant* into leaf node list. This step is same as that of original B+tree. Line 5 judges whether the key of the added item is new or not. If the key is new, its related pointers must be set. Line6-12 get pointers of new added item, and its previous item and next item. Line 13-16 set pointers of new added item. Line 18 starts the function of the second step **InsertToList**().

The second step of insertion is executed by function **InsertToList()**. The function first chooses which list should be inserted according to the operator contained in the predicate. Because three lists share the similar idea of ordering predicates according to their constants, here we introduce algorithm of inserting GreaterThan list only. The function name is **InsertToGreaterThanList()** and the algorithm is shown in Fig.7.

For line 3-4, the greaterThan node holding the item with biggest key less than or equal to Predicate.constant, so the operation here guarantee the right order of GreaterThan list. If the greaterThan node splits after insertion at line 5,

```
InsertToGreaterThanList(Sid, Predicate, LeafNode)
1  //Sid: Identifier of Subscription, Predicate: predicate with (>, >=)
2  //LeafNode: LeafNode holding the item with key  Predicate.constant
3  Insert GreaterThan list node pointed by GTPointer of the item
4         with key value Predicate.constant in the LeafNode
5  IF (GreaterThan node is split after insertion)
6     Create new node and insert it into GreaterThan list in order of key
7     Adjust GTPointers of items on leaf nodes related two nodes
8  ELSE IF (item with Predicate.constant existed before insertion)
9            Add Sid into SidSet corresponding to the item
10      ENDIF
11 ENDIF
```

Fig. 7. Algorithm of Function InsertToGreaterThanList()

line 6 inserts the new node into the list in order of key. line 7 adjusts GTPointers kept in the items of related leaf nodes by checking leaf node list according to the minimum key and maximum key kept in the items of two split GreaterThan list nodes. Line 9 adds sid into SidSet if the key of the item exited before insertion.

4.4 Search Algorithm

According to the above insertion algorithm, the searching results on GreaterThan list is the part of GreaterThan list from the item with biggest key less than input key to the Head of GreaterThan list.

```
Search(InputKey, Root, Result)
1  //InputKey: input data from event, Root: root of B+tree
2  //Result: Set of Sid and initial value is null
3  Search Inputkey from Root of B+tree
4  IF (found)
5     Assign found LeafNode to CurrentLeafNode
6     Assign found item to CurrentItem
7     Add  SideSet of CurrentItem to Result
8  ELSE  //not found
9    Assign leafNode where search(line 3) stopped to CurrentLeafNode
10 ENDIF
11 CollectPreviousINEQ(InputKey, CurrentLeafNode, Result)
12 CollectNextINEQ(InputKey, CurrentLeafNode, Result)
13 CollectGT(InputKey, CurrentLeafNode, Result)
14 CollectLT(InputKey, CurrentLeafNode, Result)
```

Fig. 8. Main Search Algorithm

The main search algorithm is shown in Fig.8. Line5-7 show the case when the *InputKey* is found. In this case, the content of SidSet kept in the found item in the leaf node should be added to the result(Line 7). Line8-10 show the case when the key is not found. In that case, the starting leaf node to get results from each list is set at Line 9. The algorithms of four **CollectPreviousINEQ()**, **CollectNextINEQ()**, **CollectGT()** and **CollectLT()** functions are similar.

In the following, only function collecting GreaterThan list(**CollectGT()**) is introduced. It is shown in Fig.9. Line 4-5 get starting node to collect results on GreaterThan list. Line 6-7 get first result item on GreaterThan list Line 8-12 collect results by scanning GreaterThan list from the first item.

```
CollectGT(InputKey, LeafNode,  Result)
1  //InputKey: input data from event
2  //LeafNode: LeafNode where B+tree search stopped.
3  //Result: Set of Sid
4  Get GTPointer of the item with the biggest key
5                           less than InputKey in LeafNode
6  Get the item which has biggest key value less than Inputkey
7      on the GreaterThan list node pointed the GTPointer(line4-5)
8  Assign the item to CurrentItem.
9  DO
10    Add SidSet of the CurrentItem into Result
11    Assign previous item of the CurrentItem to the CurrentItem
12 WHILE (Head of GreaterThan list is not met)
```

Fig. 9. Algorithm of Function CollectGT()

4.5 Delete Algorithm

The delete algorithm is a reverse procedure of insertion. Here the details are skipped for reason of space.

5 Analytical Comparison

Assume the number of unique predicate is n and the total number of predicates satisfied by event is L, the search time complexity is $O(\log n + L)$. Because pointers of the items on leaf node list need to be adjusted while do insertion as introduced in Fig.7(Line7), the number of leaf nodes accessed for adjustment of pointers is called $Number_{update}$. The minimum time complexity is $O(\log n)$ if all the pointers to be adjusted are kept in one same leaf node. Generally, the time complexity of insert operation is O(n) for the reason of adjustment. Delete operation has same time complexity for the same reason. Our algorithm is built

Table 1. Comparisons of Space and Time Complexities

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

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

Considering Complexities of space and search time, we compare with Grouped Filter[16] only for its best complexities. In publish/subscribe systems, the ratio of data arriving is much higher than that of subscriptions updating. It means that performance of search (event matching) has a decisive influence on performance of publish/subscribe system. From the view of event matching, both Grouped Filter and PB+tree have same time complexity O($\log n + L$). But as introduced in Section3, Grouped Filter uses four data structures and PB+tree use only one B+tree structure. Grouped Filter is a main memory predicate index and PB+tree is designed for secondary storage predicate index, which as far as we know, it is a novel predicate index.

Generally, the insertion and deletion complexity of PB+tree are O($\log n + number_{update}$). In practice, it is reasonable to predicate that the $Number_{update} << n$ in the case that distribution of predicates with different operators and constants is uniform. That means the performance of insert and delete is very near to minimum complexity O($\log n$) and far from maximum complexity O(n).

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 mainly support predicates with format (Constant$_{\mathrm{Start}}$ < Attribute < Constant$_{\mathrm{End}}$). In the case that the Constant$_{\mathrm{Start}}$ or Constant$_{\mathrm{End}}$ is infinite, overlap will rise greatly for IBS-tree[11] and Time Index[6], search efficiency will decline greatly for IB+tree[2]. They don't support predicate with single operator directly.

By the comparisons in table1 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 structure based on B+tree. The index structure supports predicates with relational operators(<

$, <=, =, != , >=, >$). The space complexity is O(n). The time complexity of search operation is $O(\log n + L)$, and both insertion and deletion have Minimum $O(\log n)$ and Maximum $O(n)$ time complexity. 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. Tolga Bozkaya, Meral Ozsoyoglu. Indexing transaction time database. Information Sciences 112(1998)
3. Sirish Chandrasekaran, Michael J. Franklin. Streaming Queries over Streaming Data. Proceedings of the 28th VLDB Conference, Hong Kong, 2002
4. Jiangjun Chen, David J. DeWitt, Feng Tian, Yuan Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. ACM SIGMOD 2000
5. Y.-J. Chiang and R.Tamassai, "Dynamic Algorithms in Computational Geometry". Technial Report CS-91-24, Dept. of Computer Science, Brown Univ., 1991
6. Ramez Elmasri, Gene T.J. Wuu, Yeong-Joon Kim. THE TIME INDEX: AN ACCESS STRUCTURE FOR TEMPORAL DATA. VLDB 1990
7. 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
8. Francoise Fabret, Francois Llirbat, Joao Pereira, Dennis Shasha. Efficient matching for Content-based Publish/Subscribe Systems. Technical report, INRIA, 2000.
9. 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
10. Antonin Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. ACM SIGMOD 1984
11. Eric N. Hanson, Moez Chaaboun, Chang-Ho, Yu-Wang Wang. A Predicate Matching Algorithm for Database Rule Systems. ACM SIGMOD 1990
12. Eric N. Hanson, Theodore Hohnson. Selection Predicate Indexing for Active Database Using Interval Skip List. TR94-017. CIS department, Univeristy of Florida, 1994
13. Eric N. Hanson, Chris Carnes, Lan Huang, Mohan Konyala, Lloyd Noronha. Scalable Trigger Processing. ACM SIGMOD 1999
14. Annika Hinze, Sven Bittner. Efficient Distribution-Based Event Filtering. International Workshop on Distributed Event Based Systems. Austrai July 2002
15. H.Arno Jacobsen, Francoise Fabret. Publish and Subscribe Systems. Tutorial. ICDE 2001
16. Samuel Madden, Mehul Shah, Joseph Hellerstein, Vijayshankar Raman. Continuously Adaptive Continuous Queries(CACA) over Streams. ACM SIGMOD 2002
17. Rajeev Motwani. Models and Issues in Data Stream Systems. Invited Talk. PODS 2002
18. Tak W.Yan, Hector Garcia-Molina. The SIFT Information Dissemination System. In ACM TODS 2000