A Self-Adaptive Model to Improve Average Response Time of Multiple-Event Filtering for Pub/Sub System

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, 153–8505 Japan
{botaow, zhangw, kitsure}@tkl.iis.u-tokyo.ac.jp

Abstract. Publish/subscribe system captures the dynamic aspect of the specified information by notifying users of interesting events as soon as possible. Fast response time is important for event filtering which requires multiple step processing and is also one of important factors to provide good service for subscribers.
Generally the event arrival rate is time varying and unpredictable. It is very possible that no event arrives in one unit time and multiple events arrive in another unit time. When multiple events with different workloads arrive at the same time, the average response time of multiple-event filtering depends on the sequence of event by event filtering.
As far as we know, significant research efforts have been dedicated to the techniques of single event filtering, they can not efficiently filter multiple events in fast response time. In this paper, we first propose a multiple-event filtering algorithm based on R-tree. By calculating relative workload of each event, event by event filtering can be executed with short-job first policy so as to improve average response time of multiple-event filtering. Furthermore, a self-adaptive model is proposed to filter multiple events in dynamically changing environment.
The multiple-event filtering algorithm and the self-adaptive model are evaluated in a simulated environment. The results show that the average response time can be improved maximum up to nearly 50%. With the self-adaptive model, multiple events can be filtered with average response time always same as or close to the possible best time in the dynamically changing environment.

1 Introduction

Publish/subscribe system provides 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 [7]. It captures the dynamic aspect of the specified information. Fast response time is very important for the event filtering which requires multiple-step processing, there the events need to be filtered out first as the inputs of operator like join in continuos query, and is also one of important factors to provide good service for subscribers.

Generally the event arrival rate is time varying and unpredicatable. For example, traffic monitoring, ticket reservation, internet access, stock price, weather

reports, etc.. In contrast to stable arrival rate, it's very possible that multiple events arrive in one unit time and no event arrives in another unit time.[1]

In the context of event filtering, even many index techniques such as event filtering algorithms based on multiple one-dimensional indexes [5] [8] [11] [18] [21] and event filtering algorithms based on multidimensional index [19][22], have been proposed, all these techniques are designed to filter single event instead of multiple events at one time. They can not filter multiple events directly in fast average response time if those events arrive at the same time with different workloads. Meanwhile, we found that event filtering based on multidimensional index [19] [22] is more efficient and flexible than that based on multiple one-dimensional indexes.

In order to improve average response time of multiple-event filtering, we first propose a R-tree [4] [9] based multiple-event filtering algorithm. Furthermore a self-adaptive model is proposed to filter multiple events in a dynamically changing environment with average response time always same as or close to the best possible time.

The rest of this paper is organized as follows. Section 2 introduces the background of this paper. Section 3 describes the algorithm to improve average response time. Section 4 proposes the self-adaptive model. In Section 5, the event filtering algorithm and the self-adaptive model are evaluated in a simulated environment. Section 6 discusses the related work. Finally, conclusions and future work are given out in Section 7.

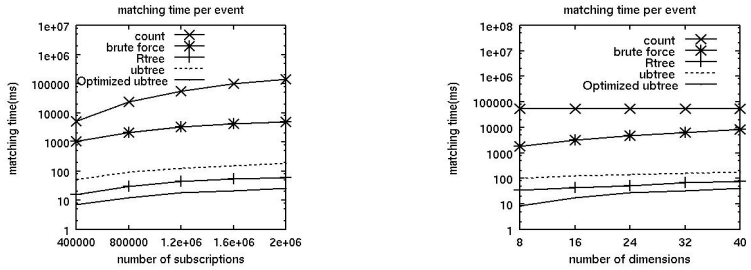
2 Background

In this section, we first explain the reason why R-tree [4] [9] is chosen, and then introduce the event filtering based on R-tree briefly. We assume that readers have enough knowledge about R-tree.

– The reasons to choose R-tree
  There are two reasons to choose R-tree here. One is performance; another is space partition strategy.
  As introduced in [19] [22], event filtering based on multidimensional index (UB-tree [2] [3] or R-tree [4] [9]) is feasible, and is much efficient and flexible than that based on Count algorithm [21], which is one representative event filtering algorithm based on multiple one-dimensional indexes. Fig.1 shows a snapshot of performance differences with two examples.[2] That's the first reason to choose R-tree.
  Further UB-tree and R-tree have different partition strategies. Generally, the search algorithms (except point query) of both index structures traverse multiple paths from root node to leaf nodes. UB-tree partitions space with space filling curve. UB-tree's search algorithm is depth-first and it is not easy to calculate the number of multiple search paths at one specified middle level. Contrary to UB-tree, R-tree decomposes the space in a hierarchical manner. Its search algorithm does not have to be depth-first, so it is easy to calculate

[1] Even logically for most of the events, there exist absolutely different arriving times, in this paper, we regard the events arriving in the same unit time as the events arriving at the same time. For example, positions reported every 30 seconds or the stock prices sampled every second.

[2] For details, please refer to [19] [22].



(a) (b)

Fig. 1. Performance Examples of Event Filtering Based on Different Index Techniques

the number of multiple search paths at one specified middle level. Because the number of multiple search paths will be used to estimate workload of event filtering in our proposal, we choose R-tree here.

– Event Filtering based on R-tree
The event filtering based on R-tree is executed as a point enclosed query. Subscriptions are hypercubes and events are points here. The dimension number means the number of attributes used in pub/sub system.
By the way, even the other multidimensional indexes, for example, multilevel grid file [20], are applicable, as the purpose of this paper is concerned, we will conentrate on the main idea to filter multiple events with unstable arrival rate in fast response time.

3 Algorithms to Improve Average Response Time

3.1 Motivation and Main Algorithm

Short-Job First(SJF) is one well-known policy used to improve average response time while scheduling multiple jobs. The critical thing is to estimate workloads properly. Meanwhile, the search algorithm of R-tree traverses multiple paths from root to leaf nodes. Apparently, the number of the multiple search paths reflects workload relatively.

Our motivation is that, in order to improve average response time of multiple-event filtering, first estimate workloads of multiple events relatively according to their numbers of search paths, and then filter these multiple events sequentially with SJF policy.

Fig.2 shows the pseudo codes of main algorithm. The algorithm is called **BatchSearch**. It is an algorithm to filter multiple events, one of whose inputs is one array of events (**EventArray**) instead of one event. The parameter **Level** controls the depth to estimate workloads starting from root node **Root**. In WorkloadTable (to be introduced later), the events will be sorted in ascending order of the number of search paths stopped at **Level**. Line 1 corresponds to workload estimation. Line 2-6 correspond to event by event filtering with SJF policy.

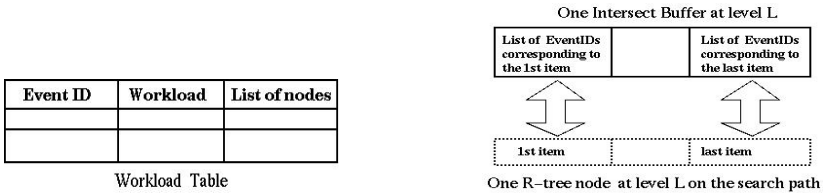
```
Begin BatchSearch(Root, EventArray, Level)
//Root. Root of R-tree
//EventArray. Multiple events arrived at the same time
//Level. Depth to estimate workload starting from the Root

1     Estimate Workloads of events in EventArray into WorkloadTable;
2     For each item in WorkloadTable;
3         Read the corresponding event and nodes located on the search paths at the input Level;
4         Search R-tree starting from the nodes obtained at line 3;
5         Output results of current event obtained at line 3;
6     ENDForLoop
End BatchSearch
```

Fig. 2. Main Algorithm BatchSearch

Line 4 filters one event with an algorithm similar to the original R-tree point enclosed query. The differences are that it starts from the nodes obtained at line 3 instead of **Root** in original point enclosed query, and the point enclosed query is executed many times (same as the number of nodes corresponding to the search paths) instead of one time. For this reason, in the following, we only describe the data structures and algorithms newly defined for workloads estimation which corresponds to Line 1 of Fig.2.

3.2 Data Structures Used to Estimate Workloads

Two data structures called WorkloadTable and IntersectBuffer, are newly defined for workloads estimation.



(a) WorkloadTable (b) IntersectBuffer

Fig. 3. Data Structures Used to Estimate Workloads

WorkloadTable is an array of items with structure shown in Fig.3-a. Each item corresponds to one event. The workload is the number of search paths(nodes) stopped at the specified **Level**. "List of nodes" are pointers of the corresponding nodes. Only one WorkloadTable is used while estimating workloads.

IntersectBuffer (Fig.3-b) is used to record events whose Minimum Bounding Rectangles (MBR) intersect with those of the items inside one R-tree node. Each level uses one intersectBuffer while estimating workloads. The items in one intersect buffer correspond to those of one R-tree node.

3.3 Algorithms to Estimate Workloads

The algorithm (corresponding to line 1 of Fig.2) to fill WorkloadTable is shown in Fig.4. In function **EstimateWorkload**, line 1 initializes the IntersectBuffer of level 0, there is only one item with one pointer pointing to the root node and all events are assumed to intersect with the MBR of this item, because the item is root. Line 2 calls a recursive procedure named **BatchIntersect** to fill the WorkloadTable. Line 3 sorts the WorkloadTable according to the workloads in ascending order.

In the procedure **BatchIntersect**, line 1-2 read the parent item of current node from the IntersectBuffer of last level (the level near to root) and get all event IDs kept in the item. Line 3 checks the ending condition of the recursive search and line 4 fills the WorkloadTable with the event IDs obtained at line 1-2 and **CurrentNode**. Line 7-16 fill the IntersectBuffer of **CurrentLevel**. Line 18-20 check next level by accessing children nodes of **CurrentNode**.

```
Begin EstimateWorkload(Root, EventArray, Level)
// Root. Root of R-tree
// EventArray. Array of events arrived at the same time
// Level. Depth to eastimate workload starting from the Root

1     Set IntersectBuffer of level 0;  // intialize intersectbuffer with all events
2     BatchIntersect(Root, EventArray, 1, Level); // Estimate from the Root
3     Sort WorkloadTable by the workload in ascending order;
End EstimateWorload

Begin BatchIntersect(CurrentNode, EventArray, CurrentLevel, Level)
//CurrentNode. The node which will be processed
// EventArray. Array of events arrtive at the same time
// CurrentLevel. The level where CurrentNode is located
// Level. Level where recursive BatchIntersect stops

1     Get parent item of CurrentNode from IntersectBuffer of CurrentLevel-1
2     Read all eventIDs into EventList from the parent item obtained at line 1;
3     IF CurrentNode is leaf node or CurrentLevel >= Level
4          Add WorkloadTable with CurrentNode and the EventList;
5          //The recursive ends here
6     ELSE
7          Reset IntersectBuffer of CurrentLevel;
8          For each item in CurrentNode // beginning of checking item
9               For each event in the EventList // beginning of checking event
10                   Check MBR intersection of current item with the current event
11                   If intersection is true
12                        Insert eventID of the event into the corresponding item in
13                             IntersectBuffer of CurrentLevel;
14                   ENDIf
15              ENDForLoop // end of checking event
16         ENDForLoop // end of checking item
17
18         For each subnode of CurrentNode
19              BatchIntersect(subnode, EventArray, CurrentLevel+1, Level);
20         ENDForLoop // End of checking subnodes
21    ENDIf
End BatchIntersect
```

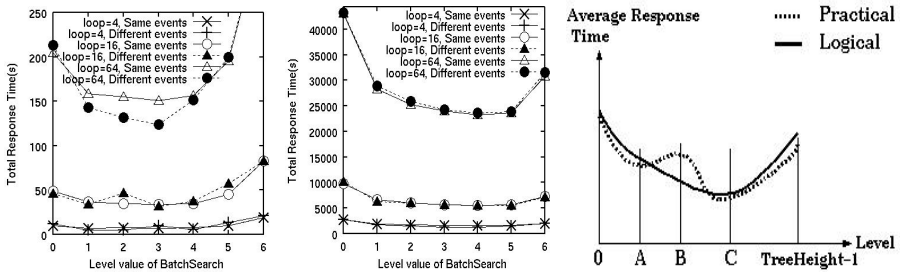
Fig. 4. Algorithms to Estimate Workload

4 Self-Adaptive Model

While filtering multiple events with **BatchSearch**, for the same multiple events, the average response time depends on the value of **Level** which controls the depth to estimate workloads. The number of multiple events arriving at the same time is not fixed, and the size and data distribution of index change dynamically also. In this section, we will propose a self-adaptive model to filter multiple events in the dynamically changing environment.

4.1 Relationship Between Average Response time and Level

Fig.5-a and Fig.5-b show two examples which reflect the relationship between average response time and **Level**. For the details of experiment environment, please refer to Section 5.1. In order to avoid overlap of results, the time here is the sum of average response times obtained with different numbers of times(loop). "Same events" means same **EventArray**s are used for different **Level**s. "Different events" means different **EventArray**s are used. The height of the index tree is 7 with 1.5 million subscriptions. The difference of two examples is the number of multiple events (size of **EventArray**).



(a) Event number=4 (b) Event number=64 (c) Changing of Average Response Time

Fig. 5. Relationship between Average Response Time and Level

The point to observe is that, with same **EventArray**s, the average response time changes in the shape of concave while level changes from root to leaf. It's reasonable considering the two main steps of **BatchSearch**: estimating workloads and event by event filtering with SJF policy. While filtering multiple events arriving at the same time, time cost to estimate workloads is overhead compared to the event filtering without workloads estimation. The overhead becomes larger with value increment of **Level**. At the same time, because the higher the **Level** is, the more accurate of the workloads estimation is. Consequently the efficiency of SJF becomes more and more higher with value increment of the **Level**. That is the reason why the average response time changes in the shape of concave.

Based on the concave, we can say that the best level exits for multiple-event filtering with **BatchSearch** if the number of events is fixed. The best level is the level to get the shortest average response time while using **BatchSearch**.

On the basis of the above observations (Fig.5-a and Fig.5-b) and analyses. The logical relationship between average response time and **Level** is expressed in Fig.5-c by the line marked **Logical**. As shown in Fig.5-a and Fig.5-b, the best level changes with different event numbers. It also depends on the size of index as shown in the evaluation (Fig.7-a).

In order to get possible best average response time, the **BatchSearch** should run with **Level** valued best in the dynamically changing environment.

4.2 Adjust Best Level Dynamically According to Statistic Information

The self-adaptive model is shown in Fig.6. The function of the self-adaptive model is to adjust the best level dynamically for multiple-event filtering in the dynamically changing environment. It is built for filtering multiple events with same event number. For multiple events with different event numbers, their statuses (current best level, numbers of updates, etc.) will be kept in different buffers.

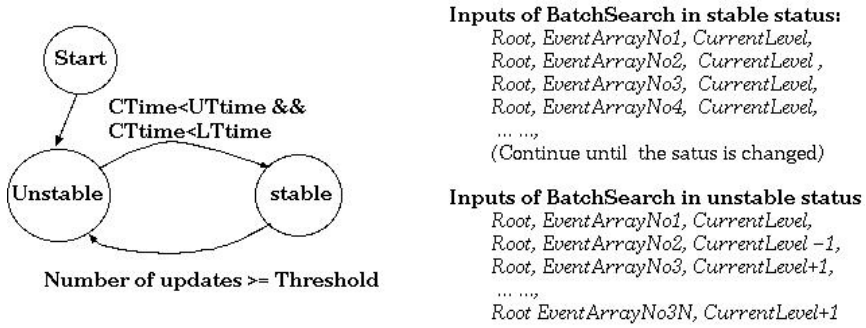


Fig. 6. Self-Adaptive Model

If the current level is best, we call system is stable. In stable status, **BatchSearch** is executed with **Level** valued best. As shown in the right of Fig.6, for arriving **EventArray**s (with same event number), same level **CurrentLevel** is used. In stable status, the average response time is the possible best time because **CurrentLevel** is best level. The number of update operations (insert and delete) is monitored and counted in stable status. After a lot of update operations, the height of the index tree or its data distribution might be changed, it is necessary to check the best level or adjust it if it changed. The system becomes unstable then. The **Threshold** shown in Fig.6 is the number to determine the time when the system enters unstable status from stable status.

Unstable status is the status in which the best level should be checked. In unstable status, the best level can be checked and gotten by trying all levels

with same **EventArray**s naively, but it's not acceptable for a dynamic system in practice. The overhead is not neglected for a higher index tree or **EventArray**s with larger size.

Our solution is that, check the average response times of current level and its upper level and its lower level (totally 3 levels), based on the "**Logical**" concave line in Fig.5-c. There, **BatchSearch** filters different **EventArray**s (same size) with **Level** values changed in a loop of round-robin way as shown in the right of Fig.6. N is the loop counter. In unstable status, multiple events are filtered with **Level** valued same as or close to the best level.

The average response times of three different levels are summed up (called CTime, UTime, LTime in Fig.6 which correspond to current level, upper level and lower level) and checked after the loop ends. Note that, the **EventArray**s are different each time and one **EventArray** is filtered just one time. If

$CTime < UTime \;\&\&\; CTime < Dtime$

is true, the system will enter stable status, because the current level is the best according to the concave changes of average response time against level value. Otherwise, adjust the current level towards to the direction of to best level (bottom of the concave line marked "**logical**", Fig.5-c) according to the concave shape and restart a new loop.

Because the contents of **EventArray**s are different, so it is possible that the average response times obtained at different levels do not change logically when the loop counter N is very small, for example, the lines marked by "Different events" with loop counter valued 4 and 16 in Fig.5-a . In this case, as expressed by the line of "**Practical**" in Fig.5-c, it is possible for system to enter stable status even the current level (A) is not best level (C). It is also possible that

$CTime > UTime \;\&\&\; CTime > LTime$

is true as shown at level (B). The self-adaptive model can not work well in these cases. But, if the value of loop counter N is larger enough, for example 64, the "**Practical**" line will change in the same concave shape or close up to "**Logical**" line statistically as shown in Fig.5-a and Fig.5-b. The loop counter is manageable for a long time running pub/sub system.

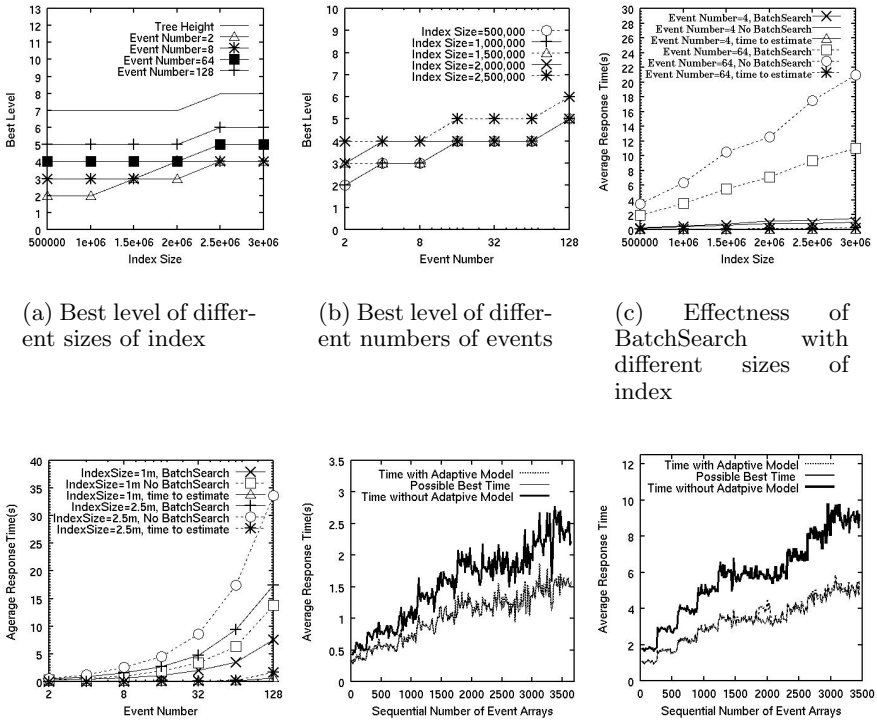
5 Results of Evaluation

5.1 Environment

The algorithm is evaluated in main memory structure. Both subscriptions and events are created randomly. The index size (number of subscriptions) changes from 0.5 million to 3.0 millions. The number of events arriving at the same time changes from 2 to 128. The **BatchSearch** algorithm is implemented on R-tree[3] with index node capacity 10 and leaf node capacity 20 in a 12D space.[4] The hardware platform is Sun Fire 4800 with 4 900MHz CPUs and 16G memory. The OS is Solaris 8.

[3] Version 0.62b. http://www.cs.ucr.edu/ marioh/spatialindex

[4] The performance doesn't change drastically if the dimension number is located in a reasonable range as shown in Fig.1. Dimension number and node capabilities influence the performance of R-tree itself but do not influence the improvement of average response time and effectiveness of the self-adaptive model which are mainly concerned in this paper.



(a) Best level of different sizes of index

(b) Best level of different numbers of events

(c) Effectness of BatchSearch with different sizes of index

(d) Effectness of BatchSearch with different numbers of input

(e) Performance of unstable status (event number=8)

(f) Performance of unstable status (event number=32)

Fig. 7. Evaluation Results of **BatchSearch** and Adaptive Model

5.2 Evaluation of BatchSearch Algorithm

Changing of Best Level Fig.7-a shows that the best level changes slowly with increment of index size. It means the **Threshold** in Fig.6 can be set larger, for example 100,000, if the insert operation is more frequent than delete operation. For pub/sub system with balanced insert and delete operations, the value of **Threshold** is implementation-dependent. Generally, the update operations are much less than filtering operation. So in most of time, system can run in stable status. Fig.7-b shows that the smaller the number of events is, the lower the best level is.

Improvement of Average Response Time Fig.7-c and Fig.7-d compare the average response time of **BatchSearch** with **Level** valued best to that without considering about workloads ("no BatchSearch". **BatchSearch** is not used, multiple events are filtered event by event with original point enclosed

query in a random sequence). Fig.7-c shows that the improvement of average response time has good scalability with increment of index size. Fig.7-d shows that the larger the number of events is, the more the average response time can be improved. The reason is that for the events with uniform distribution of workloads, the larger the number of events is, the more the SJF can be benefited. The maximum improvement is nearly up to 50% in our evaluation. Both Fig.7-c and Fig.7-d also show that the cost to estimate workload (algorithms shown in Fig.4) can be neglected compared to the improvement of average response time.

Effectiveness of Self-Adaptive Model Fig.7-e and Fig.7-f compare the performance with the self-adaptive model to that without the self-adaptive model (same as "no BatchSearch" in Fig.7-c and Fig.7-d) and the possible best performance. There, the size of index changes from 0.5 million to 2.6 millions, the **Threshold** is 300,000, and the loop counter is 64. When the system becomes stable, 300,000 subscriptions are inserted into the index. So Fig.7-e and Fig.7 show the performance of unstable status. The difference is the number of events.

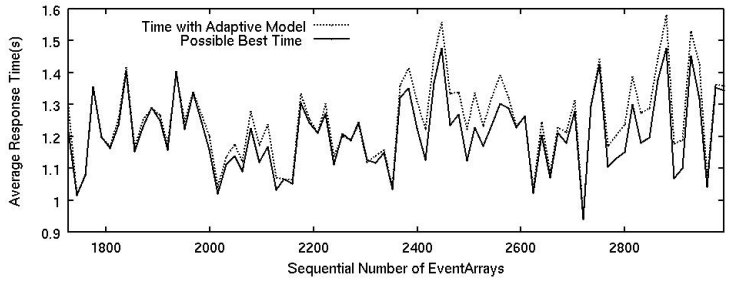


Fig. 8. One Piece of Unstable Status Performance (event number=8, index size=2,000,000)

We can find that the average response time with the self-adaptive model is much better than that without the self-adaptive model (**BatchSearch** is not used), the time differences are almost at the same level as those shown in Fig.7-c and Fig.7-d which are obtained with best level. Even in unstable status, the time obtained by using self-adaptive model is very close to the possible best time as shown in Fig.7-e and Fig.7-f. The time difference compared to the possible best time is so small that is hard to make difference in Fig.7-e and Fig.7-f. Fig.8 shows a piece of details of Fig.7-e where index size is 2 millions (the range of sequential number is about 1550-3000).

We can say that with the self-adaptive model, multiple events can be filtered with average response time same as or close to the possible best time.

6 Related Work

A lot of algorithms related to event filtering have been proposed. They are proposed for publish/subscribe systems [1] [8] [12] [18] [19] [21] [22], for continuous queries [5] [6] [15] and for active database [10] [11].

Predicate indexing techniques have been widely applied. There, a set of one-dimensional index structures are used to index the predicates in the subscriptions. Mainly, there are two kinds of multiple one-dimensional indexs based algorithms: Count algorithm [21] and Hanson algorithm [10] [11]. The performances of Count algorithm and Hanson algorithm have same complexity order, they differ from each other by whether or not all predicates in subscriptions are placed in the index structures. Meanwhile in [19] [22], event filtering based on multidimensional index is proved to be feasible and efficient compared to the popular Count algorithm. The conclusions of [19] [22] are the basis of this paper.

The testing networking based techniques [1] [12] initially preprocess the subscriptions into a matching tree. Different from predicate index, [1] and [12] built subscription trees based on subscription schema. They suffer from the problem of space and maintenance.

Event filtering is one critical step of continuous queries. In [5], predicate index is built based on Red-Black tree, there algorithm is similar to bruteforce which scans the total Red-Black tree every time when event arrives. In [6], Count algorithm is used. Adaptivity is applied in [15], it implements routing policies to let faster operators filter out some tuples before they reach the slower operators. In [17], queries are optimized based on rate of input to minimize response time by introducing event arrival rates into the optimizer cost model.

As far as we know, the problem of adaptively improving average response time for multiple events arriving at the same time has not been addressed yet.

7 Conclusions and Future Work

In this paper, in order to improve the average response time of pub/sub system with unstable event arrival rate, we first proposed a multiple-event filtering algorithm based on R-tree. The relative workload of each event is estimated according to the number of search paths so as to utilize short-job first policy. Further a self-adaptive model is designed to filter multiple events in dynamically changing environment.

According to the evaluation results, the improvement of average response time has good scalability with index size and the larger the number of events is, the more the average response time can be improved. The average response time can be improved maximum up to nearly 50%. The results also show that the overhead derived from workloads estimation can be neglected compared to the improvement of average response time. With the self-adaptive model, multiple events can be filtered with average response time always same as or close to the possible best time.

Because the proposed idea and self-adaptive model can be applied to other multidimensional index structure also, for example, multilevel grid file [20], in the future, we will try other applicable multidimensional indexs in different update scenarios and real data.

References

[1] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, T. D. Chandra. Matching Events in a Content-based Subscription System. Eighteenth ACM Symposium on Principles of Distributed Computing(PODC), 1999:53-61

[2] R. Bayer. The Universal B-Tree for multidimensional Indexing. Technical Report TUM-I9637, November 1996

[3] R. Bayer, V. Markl. The UB-Tree: Performance of Multidiemnsional Range Queries. Technical Report TUM-I9814, June 1998

[4] N. Beckmann, H.-P. Kriegel, Ralf Schneidar, Berhhard Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. SIGMOD 1990:322-331

[5] S. Chandrasekaran, M. J. Franklin. Streaming Queries over Streaming Data. Proceedings of the 28th VLDB Conference, Hong Kong, 2002:203-214

[6] J. Chen, D. J. DeWitt, F. Tian, Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. ACM SIGMOD 2000:379-390

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

[8] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, D. Shasha. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems. ACM SIGMOD 2001:115-126

[9] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. ACM SIGMOD 1984:47-57

[10] E. N. Hanson, M. Chaaboun, C.-H., Y.-W. Wang. A Predicate Matching Algorithm for Database Rule Systems. ACM SIGMOD 1990:271-280

[11] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha. Scalable Trigger Processing. ICDE 1999:266-275

[12] A. Hinze, S. Bittner. Efficient Distribution-Based Event Filtering. International Workshop on Distributed Event Based Systems. Austrai July 2002:525-532

[13] H. A. Jacobsen, F. Fabret. Publish and Subscribe Systems. Tutorial. ICDE 2001

[14] V. Markl. MISTRAL:Processing Relational Queries using a Multidimensional Access Tecnnique. Ph.D. Thesis, TU Munchen, 1999, published by infix Verlag, St.Augustin. DISDBIS 59, ISBN 3-89601-459-5, 1999

[15] S. Madden, M. Shah, J. Hellerstein, V. Raman. Continuously Adaptive Continuous Queries(CACA) over Streams. ACM SIGMOD 2002:49-60

[16] F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, R. Bayer. Intergrating the UB-tree into a Database System Kernel. VLDB 2000:253-272

[17] S. D. Viglas, J. F. Naughton. Rate-based query optimization for streaming information sources. SIGMOD Conference 2002: 37-48

[18] B. Wang, W. Zhang, M. Kitsuregawa. Design of B+Tree-Based Predicate Index for Efficient Event Matching. APWeb 2003: 548-559

[19] B. Wang, W. Zhang, M. Kitsuregawa. UB-Tree Based Efficient Predicate Index with Dimension Transform for Pub/Sub System. DASFAA 2004: 63-74

[20] K.Y. Whang, R. Krishnamurthy. The Multilevel Grid File - A Dynamic Hierarchical Multidimensional File Structure. DASFAA 1991:449-458

[21] T. W. Yan, H. Garcia-Molina. The SIFT Information Dissemination System. In ACM TODS 24(4):529-565 1999

[22] W. Zhang. PERFORMANCE ANALYSIS OF UB-TREE INDEXED PUBLISH/SUBSCRIBE SYSTEM. Master Thesis. Department of Information and Communication Engineering, The University of Tokyo. March 2004