社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

複数入力を持つ拡張R-tree検索アルゴリズムを用いたPUB/SUBシステムの平均応答時間の改善

王　波涛†　　張　旺†　　喜連川　優†

† 東京大学　生産技術研究所　〒153–8505 東京都目黒区駒場 4–6–1
E-mail: †{botaow,zhangw,kitsure}@tkl.iis.u-tokyo.ac.jp

あらまし　Publish/subscribe システムはユーザに対して興味があるイベントを随時送信している。一般的に、イベントが起こる確率は時間によって変化し、予測ができない。一定の時間内にイベントが何も起こらなかったり、複数のイベントが同時に発生することは良く起こる。複数のイベントが同時に発生した時、その平均レスポンスタイムは作業の処理順序に依存する。本論文は始に R-tree を用いた複数イベントのためのフィルタリングアルゴリズムを提案する。各々のイベントの負荷に関する情報を用いることで、処理時間が短いイベントから処理を行い、平均レスポンス時間を向上させる。さらに、インデックスが動的に変化する環境下で、サイズの異なるイベント集合のための自己適応モデルの提案と評価を行う。

キーワード　Publish/Subscribe, Event Filtering, Multiple Inputs, R-tree

Adaptively Improving Average Response Time of Pub/Sub System Based on Extended R-Tree Search Algorithm with Multiple Inputs

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

† Institute of Industrial Science, The University of Tokyo
Komaba 4–6–1, Meguro-ku, Tokyo, 153–8505 Japan
E-mail: †{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. Generally the rate of event arriving is time varying and unpredictable. It is very possible that no event arrives in an unit time and multiple events arrive in another unit time. When multiple events arrive at same time, the average response time of events filtering depends on the sequence of filtering events which have different workloads. In this paper, we first propose an event filtering algorithm with multiple inputs (multiple events) based on R-tree. With information of relative workload of each event, event by event filtering can be executed with short-job first policy which improves average response time of multiple jobs. Further a self-adaptive model is proposed and evaluated to filter set of events with different sizes on dynamically changed index.

Key words　Publish/Subscribe, Event Filtering, Multiple Inputs, R-tree

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. Efficient event filtering with a faster response time is very important for event processing with multiple steps like join, and is one of important factors to provide good service for subscribers.

Generally the rate of event arriving is time varying and unpredicatable. For example, traffic monitoring, ticket reservation, internet access, stock price, ect.. In contrast to stable rate, it's very possible that a batch of events arrive in one unit time and no event arrives during another unit time.(注1)

In the context of publish/subscribe system, even many index techniques such as multiple one-dimensional indexes based [5] [8] [10] [14] [16] and multidimensional index based [15] have been proposed for event filtering, all these techniques are designed to filter events one by one. They can not deal directly with batch events in a fast average response time if those events arrive at same time with different workloads. Meanwhile, we found that event filtering based on multidimensional index [15] [17] is more efficient and flexible

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

than that based on multiple one-dimensional indexes. In order to improve average response time of event filtering in above case, in this paper, we first propose a R-tree [4] [9] based event filtering algorithm with multiple inputs which are events arriving at same time. A cost model to estimate relative workloads of these events is built to arrange the filter order of these events with Short-Job First (SJF) policy. Further, because the **input size** representing the number of events arriving at the same time and **index size** representing the number of subscriptions kept in index, change dynamically, an adaptive model is proposed to filter events with average response time always same as or close to the possible best time.

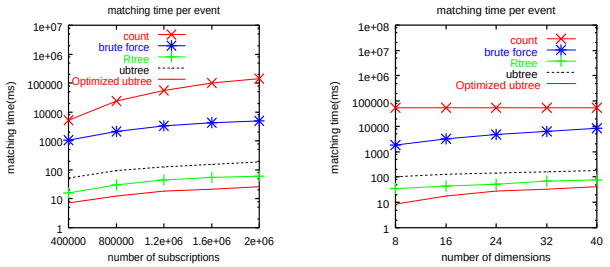
The rest of this paper is organized as follows. Section 2 introduces the background and motivation. Section 3 introduces the algorithm to improve average response time. Section 4 proposes the adaptive model. In Section 5, the event filtering algorithm and the adaptive model are evaluated and analyzed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

2. Background and Motivation

As introduced in [15] [17], multidimensional index (R-tree [4] [9] or UB-tree [2] [3]) based event filtering is feasible and is much efficient and flexible than that based on the popular multiple one-dimensional indexs based technique - count algorithm [16]. Fig.1 shows a snapshot of performance difference with two examples.(注2)

Meanwhile, SJF is one well-known policy used to improve average response time while scheduling multiple jobs. The critical thing is to estimate workloads correctly.

Even UB-tree and R-tree have different partition strategies, the search algorithms of both index structures traverse multiple paths from root node to leaf nodes. Apparently, the number of the multiple search paths reflects workload relatively. Our motivation is that make use of this property to estimate workload of each event so as to improve average response time of events filtering with SJF policy in the case that events arriving at same time.



(a) (b)

図 1 Performance examples of event filtering based on different index techniques

Because UB-tree partitions space with space filling curve, original UB-tree's search algorithm is depth-first and no Minimum Bounding Rectangle (MBR) information is required and kept inside its nodes, it's not easy to use MBR of event to calculate the number of multiple search paths at specified middle levels of original UB-tree without accessing leaf nodes. The structure of R-tree doesn't have this problem. For this reason, we choose R-tree as the basis of our proposal in this paper.

(注2)：For details, please refer to [15] [17].

3. Improve Average Response Time

3.1 Basic Idea and Main Algorithm

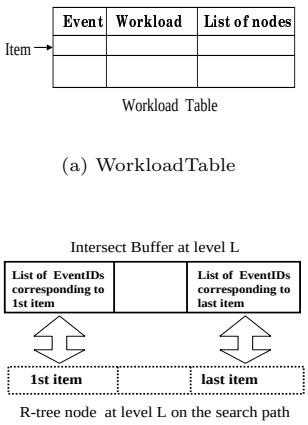
The basic idea is that for the events arriving at same time, the relative workloads are estimated respectively first based on their different numbers of search paths on R-tree and then do filtering event by event with SJF policy with assumption that the more the number is, the higher the workload is. Fig.2 shows the pseudo codes of the main algorithm. The

```
Begin BatchSearch(Root, EventArray, EventNumber, Level)
1   Estimate Workloads of EventArray into WorkloadTable;
2   For each item in WorkloadTable;
3     Read event and nodes located on the search paths at the Level;
4     Search R-tree starting from the nodes with the event;
5     Output results of current event;
6   ENDForLoop
End  BatchSearch
```

図 2 Main Algorithm-BatchSearch

algorithm is called **BatchSearch**. The input includes an array of events **EventArray** and its size **EventNumber**. The parameter **Level** controls the depth to estimate workload starting from root node **Root**. Line 1 estimates and sorts workload by checking all input events simultaneously to level **Level**. In WorkloadTable (to be introduced later), the events are sorted in ascending order of the number of search paths at level **Level** and the nodes on the search paths are kept in the corresponding item of WorkloadTable. Line 4 does event filtering (search of R-tree) with algorithm similar to original R-tree search algorithm except that the start point is not **Root** but the nodes gotten at line 3. The results of each event are outputted immediately at line 5 without waiting for the ending of the filtering of last event even they arrived at same time.

3.2 Data Structures



(a) WorkloadTable

(b) IntersectBuffer

図 3 Data Structures Used in BatchSearch

WorkloadTable is an array of items with structure shown

in Fig.3-a. Each item corresponds to one event of **EventArray**. The Workload in Fig.3-a is the number of nodes located at the ending of search paths stopped at the specified **Level**. WorkloadTable is filled and sorted at line 1 of Fig.2 by function named **EstimateWorkload**. A data structure named IntersectBuffer (Fig.3-b) is used to record events whose MBRs intersect with those of items of one R-tree node which is located in the paths from root to the specified level. Because the algorithm to estimate workload is similar to the depth-first search of R-tree, at any time while estimating workload, each level corresponds to only one intersect buffer.

The number of items in one intersect buffer is same as that of one R-tree node. The 1st item of intersect buffer corresponds the 1st item of R-tree node. The content of the 1st item of IntersectBuffer is the list of eventIDs whose MBRs intersect with that of the 1st item of the R-tree node. The others items have similar contents.

```
Begin EstimateWorkload(Root, EventArray, EventNumber, Level)
1  Set IntersectBuffer of level 0;
2  BatchIntersect(Root, EventArray, 1, Level);
3  Sort WorkloadTable in ascending order;
End EstimateWorload

Begin BatchIntersect(CurrentNode, EventArray, CurrentLevel, Level)
1  Get the item which CurrentNode corresponds to from IntersectBuffer
2        of CurrentLevel-1 and read all eventIDs into EventList;
3  IF CurrentNode is leaf Node or CurrentLevel  >= Level
4     Add WorkloadTable with CurrentNode and EventList;
5  ELSE
6      Reset IntersectBuffer of CurrentLevel;
7      For each item in CurrentNode
8        For each event in EventList
9           Check MBR intersection of current item with current event
10          If intersect
11             Insert eventID into the corresponding item of
12                   IntersectBuffer of CurrentLevel;
13          ENDIf
14        ENDForLoop
15     ENDForLoop
16     For each item of CurrentNode
17       BatchIntersect(Item's SubNode, EventArray, CurrentLevel+1, Level);
18     ENDForLoop
19 ENDIf
End BatchIntersect
```

図 4　Algorithm of Estimating Workload

The algorithm to fill WorkloadTable is shown in Fig4. In function **EstimateWorkload**, line 1 intializes the IntersectBuffer of level 0 where there is only one item with one pointer pointing to the **Root** node and all events are assumed to intersect with MBR of this item. The SJF can not be benefited by **BatchSearch** with **Level** valued 0, because all workloads have same value 1. Line 2 call a recursive function **BatchIntersect** to fill WorkloadTable, level 1 means checking from root node. Line 3 sorts WorkloadTable according to the number of search paths in ascending order.

In function **BatchIntersect**, line 1-2, read one item from IntersectBuffer of last level (the level nearer to root) and gets all event IDs kept in the item. That item corresponds to the item of R-tree node at last level which includes the pointer pointing to **CurrentNode**. Line 3 checks the ending condition of recursive search and line 4 adds the WorkloadTable with the event IDs gotten at line 1-2 and **CurrentNode**. Line 6-15 fill intersect buffer of **CurrentLevel**. Line 16-18 search next level by accessing subnodes of **CurrentNode**.

4. Model of Adaptive Search

For same batch events, the performance of **BatchSearch** depends on the value of **Level**. At the same time, the number of events arriving at same time is not fixed, the size of index changes dynamically also. In this section, we will propose a self-adaptive model in order to filter kinds of events with average response time same as or close to the possible best time.

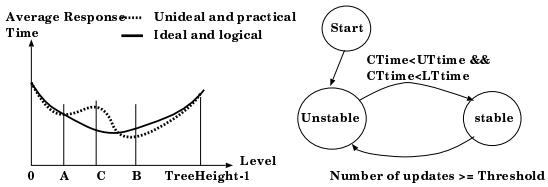


図 5　Changing of Average Response Time and Adaptive Model

4.1 Performance Analysis

While dealing with events arriving at same time, line 1 of main algorithm **BatchSearch** is a kind of overhead compared to the processing with original R-tree search algorithm event by event without workload estimation. The overhead becomes larger with the value increment of **Level**. At the same time, because the higher the **Level** is, the more accurate of workload estimation is, the efficiency of SJF become more and more better with the value increment of **Level** also. For the same batch events, the average response time based on **BatchSearch** is a function of **Level** L. Their relationships can be described in the follows:

$$ARTime(L) = ETime(L) + TETime/Efficiency(L)$$

$ARTime$ is the average response time. $ETime$ is the time to estimate workload. $TETime$ is the total execution time of processing all events. It is a constant related to the total workloads of all events and doesn't change with **Level**. $Efficiency$ is the efficiency of SJF. It's the ratio of response time gotten at leaf level to the response time gotten at level L under policy of SJF using number of paths to estimate relative workloads. The larger it is, the shorter the averager response time is. Because it has shape of concave as shown on the leaf of Fig.5 with mark "**Ideal and logical**". The best level exits for the batch events with same event number and it should be located between level 0 to level TreeHeight-1. The best level changes for different number of input events and size of index.

In order to get best average response time, the **BatchSearch** should run with **Level** valued best level.

4.2 Adjust Best Level Dynamically According to Statistic Information

The adaptive model is shown on the right of Fig.5, it is built for filtering of batch events with same event number arriving at same time. For events with different sizes (numbers), their statuses will be kept in different buffers, for example, entries of a status buffer array for batch events with different sizes. The main function of the model is to adjust best level dynamically for filtering of different batch events with different numbers and sizes of index.

If the current level is best level, we call the system is stable. In stable status, the **BatchSearch** is executed with **Level** valued best level. The number of update operations (insert and delete) is monitored in stable status. After some update operations, the height of index tree or data distribution of index maybe be changed, it is necessary to check the best level or adjust it if it changes. The system becomes unstable then. The **Threshold** shown in Fig.5 is the number to determine the time when the system enters unstable status from stable status.

In unstable status, the best level can be checked by trying all levels with same events naively, but it's not acceptable for an dynamic system in practice. This overhead is not neglected for a higher index tree or batch events with larger number.

Our model is that check current level and its upper and lower levels (totally 3 levels) based on the "**Ideal and logical**" changes of **BatchSearch** performance. In unstable status, for events arriving batch by batch with same size at different time, **BatchSearch** process these batches of events with **Level** value changed in a way of round-robin loop. The input contents of **BatchSearch** change in the sequence of arriving time like

$(EventArrayNo1, CurrentLevel)$,
$(EventArrayNo2, CurrentLevel - 1)$,
$(EventArrayNo3, CurrentLevel + 1)$, ...,
$(EventArrayNo3N, CurrentLevel + 1)$.

N is the counter of loop. So in unstable status, system does events filtering with **Level** values same as or close to the best level.

Besides filtering these batchs of events in unstable status, the average response times of three different levels (called CTime, UTime, LTime in Fig.5 which correspond to the average response time of current level, upper level,lower level) are summed up and checked when the loop ends. If $CTime < UTime \;\&\&\; CTime < Dtime$ is true, the system will enter stable status, because the current level is the best level for **BatchSearch** based on the "**Ideal and logical**" changes of **BatchSearch**. And the current update operation counter is recorded for the update checking in stable status. Otherwise, moves the current level towards to the direction of best level according to the "**Ideal and logical**" changes of **BatchSearch** performance and restarts a new loop.

Because for every events filtering , the input **EventArray** of **BatchSearch** is different, so it's possible that the time gotten at different levels doesn't change "ideally and logically" when the loop counter N is very small, for example 1. In this case as the line of "**Unideal and practical**" shown in Fig.5, it is possible for system to enter stable status, even the current level(A) is not best level(B). It is also possible that $CTime > UTime \;\&\&\; CTime > LTime$ is true as shown at level(C). The adaptive model can not work well in this case. But, if the value of loop counter N is bigger enough, the "**Unideal and practical**" line will change in the same concave shape or close up to "**Ideal and logical**" line statistically. The adaptive model is expected to work well if the loop number is big enough, It's managable for a long time running pub/sub system.

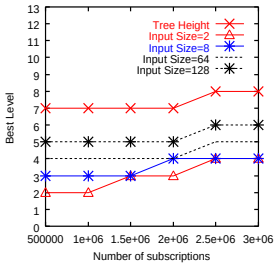
When system starts, the status is unstable. The current level can be set with any value allowable, for example TreeHeight/2. At last, the system will become stable, because the filtering operation is more frequent than the update operation in pub/sub system and the best level changes slowly with increment of index size shown in later evluation.

5. Results of Evaluation

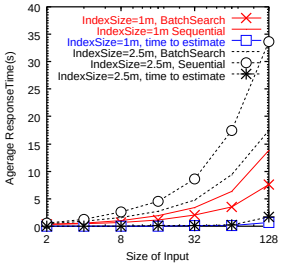
5. 1 Environment

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

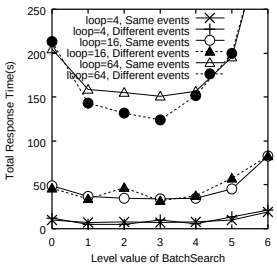
5. 2 Evaluation of BatchSearch Algorithm



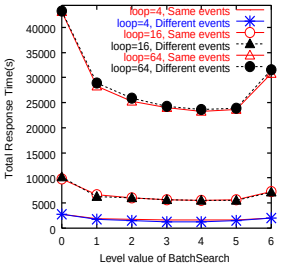
(a) Best level of different size of index



(b) Effectness of BatchSearch with different size of input



(c) Loop counter and best level (input size =4)



(d) Loop counter and best level (input size=64)

図 6 Evaluation Results of **BatchSearch** Algorithm

The evaluation results are shown in Fig.6. Fig.6-a shows that the best level changes slowly with increment of index size. It means the **Threshold** of Fig.5 can be set very large, for example 100,000 in the case that insert operation is more frequent than delete operation.

Fig.6-b compare the average response time of **BatchSearch** algorithm to that which just inputs events to original R*-tree in a random sequence. Further, it shows that the cost to estimate workload (algorithms shown in Fig.4 can be neglected compared to average response time. It also shows that the larger the size of input is, the more the average response time can be improved. The reason is that for the events with uniform distribution of workloads, the the larger the size of input is, the more the SJF can be benefited. The maximum is nearly up to 50% in our evaluation.

Fig.6-c and Fig.6-d show the changing of response time which are calculated with same and different events gotten at different levels. The input with different events is used to simulate the performance in unstable status. There the index size is 1.5 million and the height of tree is 7. The difference of Fig.6-c and Fig.6-d is the input size. Boths figures show that the smaller the loop counter and input size are, the bitterly the time based on different events changes on different levels. They also show that with increment of loop counter, the trembling of response time "**Unideal and practical**"

(注3) : The performance doesn't change drastically if the number of dimension is located in a reasonable range as shown in Fig.1

(注4) : Version 0.62b. http://www.cs.ucr.edu/ marioh/spatialindex

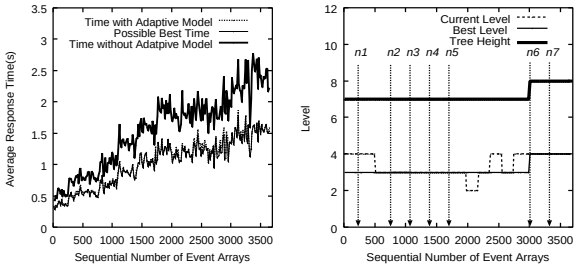
which is introduced in Fig.5 and captured clearly in Fig.6-c (loop=4 or loop=16, Different events), becomes more and more stable with shape of ideal concave and merge into the line gotten with the same events at each level.

5.3 Evaluation of Adaptive Model

Logically, the performance of stable status is better than that of unstables stable if the **Level** of **BatchSearch** function is the best level. The effectness of the adaptive model in stable status has been shown in Fig.6-c and Fig.6-d in last section. This section mainly shows the results related to the performance of *unstable status* and the best level when the system becomes stable status.

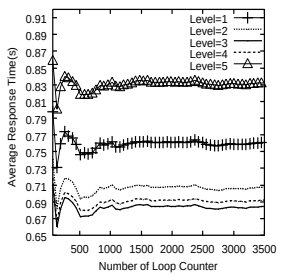
Fig.7-a shows the performance of unstable status compared to the performance without using the adaptive model (same as the "No BatchSearch" performances shown in Fig.6-b) and the possible best performance. There, the size of index changes from 0.5 million to 2.6 millions, the **Threshold** is 300,000, and the loop counter is 64. The initial value of current level is $TreeHeight/2+1$. The $TreeHeight$ is 7 when the index size is 0.5 million. When the system becomes stable, 300,000 objects (subscriptions) are inserted into the index. So Fig.7-a show the performance of unstable status.

We can find that performance with the adaptive model is much better than the performance without the adaptive model, the performance differences are almost at same level as those shown in Fig.6-b which are gotten at stable status. The performance of the adaptive model even in unstable status is very close to the possible best performance as shown in Fig.7-a We can say that with the adaptive model, the arrays of events can be filtered with response time close to the possible best time. The difference can be neglected compared to the performance without adaptive model.

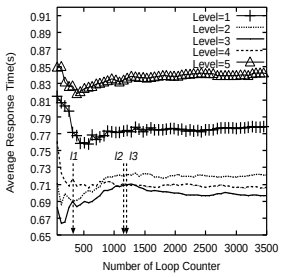


(a) Performance of unstable status (input size=8)

(b) Changing of current depth(input size = 8)



(c) Performance of Different Levels (same events)



(d) Performance of Different Level (different events)

図 7 Results of Adaptive Model Evaluation

Corresponding to Fig.7-a, fig.7-b shows the changing of current level while system changes from unstable status to stable status. The dash lines labeled n1, n2, n3, n4, n5, n6 and n7 are points in stream of input event arrays which are input and filtered sequentially. There the system becomes stable with current index size and enters unstable status after inserting 300,000 objects (subscriptions). It shows that the current level becomes same as the best level when system enter stable status at points n2, n3, n4, n5 and n7 in Fig.7-b. But, at points n1 and n6, the current level values (both are 4) are different from the best level value (3). It means that the **Level** of **BatchSearch** function isn't the best depth. Fig.7-c shows the performance difference compared to the possible best performance in this case. Fig.7-d shows the reason why nonbest level is gotten possibly when system becomes stable status.

The difference of Fig.7-c and Fig.7-d is that whether the same event arrays or different arrays are applied to the different levels close to the best level. The index size is 1.1 millions. Loop counter is the loop counter defined in Section 4.2. The average response time are calculated on accumulating total response time and the value of its corresponding loop counter. The Fig.7-c shows that the performance of level 3 (best level) are better and close to those of two neighbor levels, level 2 and level 4. Especially, the performance of level 4, which is the current level when system become stable at points n1 and n6 shown in Fig.7-d, is very close to that of level 3.

From Fig.7-d, we can find that the reasons why the current level is different from the best level when system becomes stable are that 1) the performance of the best level is very close to those of its two neighbor levels, and 2)the value of loop counter is too smaller.

For example, in Fig.7-c, the performances of levels 2, 3 and 4 are 0.708093s, 0.684761s and 0.692080s when the loop counter is 2000, there the loop counter is large enough to capture the representative performance. The performance of level 3 and 4 is less than 1.07%. At the same time, as shown in Fig.7-d, where different event arrays are filtered at these three different levels, the performance of the best level is not always smaller than those of its two neighbor levels when the loop counter is smaller (less than 2000).

In Fig.7-d, at point of $l1$ (Loopcounter = 320), the performance of levels 2, 3, 4 are 0.689795s, 0.689983s, 0.716901s, there the performance of level 2 is better that of best level 3, and at point of $l2$ and $l3$ (Loopcounter=1216 and 1280) are 0.722000s, 0.709347s,0.709328s and 0.721254s, 0.710612s,0.708728s where the performances of level 4 are better than that of best level 3. In summary, when the counter loop is smaller, the performances of different levels change a little bitterly. The possibility is high for the two adjacent and close performance lines to intersect each other.

That's reason why the current level (shown in Fig.7-b, interval between n5 and n6) changes closely to the best level when index size is 2,000,000, the loop counter is 64 there. But when loop counter become larger, for example 2000, the Fig.7-c and Fig.7-d have same and stable performance difference.

Because the loop counter is managable in practice, we can say that if the loop counter is larger enough, the system will enter stable status with the best level or the level (in the case that loop counter is not larger enough) with performance very close to that of the best level.

6. Related Work

A lot of algorithms related to event filtering have been

proposed. They are proposed for publish/subscribe systems [1] [8] [11] [14] [15] [16], for continuous queries [5] [6] [12] and for active database [10].

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 predicate indexing based algorithms: counting algorithm [16] and Hanson algorithm [10]. They differ from each other by whether or not all predicates in subscriptions are placed in the index structures. [16] is an Information Dissemination System (IDS) for document filtering.

In [14] [15], multidimensional index based event filtering is proved to be feasible and efficient. It's the basis of this paper.

The testing networking based techniques initially preprocesses the subscription into a matching tree. Different from predicate index, [1] and [11] built subscription trees based on subscription schema. For the reason of space and maintenance, they are impractical for pub/sub application.

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 that scans the Red-Black tree for event filtering each time. [12] implemented routing policies to let faster operators filter out some tuples before they reach the slower operators. In [13], queries are optimized based on rate of input.

The problem of multiple events arriving at same time with different workloads is not considered in above techniques.

7. Conclusion and Future Work

In this paper, for pub/sub system, we first proposed an event filtering algorithm with multiple events as input based on R-tree. The relative workload of each event is estimated according to the number of search paths. Short-Job First policy is utilized to improve the average response time. Further an adaptive model is designed to filter multiple events arriving at the same time with average response time always close to the best time for different input sizes and changing index size. According to the evaluation results, the average response time can be improved maximumly up to nearly 50% with our algorithm and the adaptive model can work well with average response time same as or close to the possible best time in both stable and unstable statuses.

In the future, we will evaluate adaptive model with real data in different update scenarios and data skew related to index structure.

文　　献

[1] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 53–61. ACM Press, 1999.

[2] R. Bayer. The universal b-tree for multidimensional indexing. Technical Report TUM-I9637, Technische Universitat Munchen, November 1996.

[3] R. Bayer and V. Markl. The ub-tree: Performance of multidimensional range queries. Technical Report TUM-I9814, Technische Universitat Munchen, June 1998.

[4] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 322–331. ACM Press, 1990.

[5] S. Chandrasekaran and M. J. Franklin:. Streaming queries over streaming data. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases*, pages 203–214. ACM Press, 2002.

[6] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: a scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 379–390. ACM Press, 2000.

[7] P. T. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. Technical Report Technical Report DSC ID:2001, Swiss Federal Institute of Technology, January 2001.

[8] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 115–126. ACM Press, 2001.

[9] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD'84, Proceedings of Annual Meeting*, pages 47–57. ACM Press, 1984.

[10] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. B. Park, and A. Vernon. Scalable trigger processing. In *Proceedings of the 15th International Conference on Data Engineering, 23-26 March 1999, Sydney, Austrialia*, pages 266–275. IEEE Computer Society, 1999.

[11] A. Hinze and S. Bittner. Efficient distribution-based event filtering. In R. Wagner, editor, *22nd International Conference on Distributed Computing Systems (ICDCS- 2002), Workshops: 1st International Workshop on Distributed Event-Based Systems(DEBS)*, IEEE Computer Society, 2002.

[12] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 49–60. ACM Press, 2002.

[13] S. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 37–48, 2002.

[14] B. Wang, W. Zhang, and M. Kitsuregawa. Design of b+tree-based predicate index for efficient event matching. In *Web Technologies and Applications, 5th Asian-Pacific Web Conference, APWeb 2003, Xian, China, April 23-25, 2003, Proceedings*, pages 537–547, 2003.

[15] B. Wang, W. Zhang, and M. Kitsuregawa. UB-Tree based efficient predicate index with dimension transform for pub/sub system. In *Database Systems for Advances Applications, 9th International Conference, DASFAA 2004, Jeju Island, Korea, March 17-19, 2004, Proceedings*, pages 63–37, 2004.

[16] T. W. Yan and H. Garcia-Molina. The sift information dissemination system. *ACM Trans. Database Syst.*, 24(4):529–565, 1999.

[17] W. Zhang. Performance analysis of Ub-tree indexed publish/subscribe system. Master's thesis, Department of Information and Communication Engineering, The University of Tokyo, 2004.