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

Botao WANG♠ Wang ZHANG♦
Masaru KITSUREGAWA♠

Publish/subscribe system captures the dynamic aspect of the specified information by notifying users of interesting events as soon as possible. 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 an extended R-tree search algorithm. With information of relative workload of each event, event by event filtering can be executed with short-job first policy to improve average response time of multiple events filtering. Further a self-adaptive model is proposed to filter multiple events with average response time always same as or close to the possible best time in dynamically changing environment.

1. Introduction

Efficient event filtering with a faster response time is very important for event filtering which requires multiple step processing like event join, and is also 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, etc. In contrast to stable arrival rate, it's very possible that multiple events arrive in one unit time and no event arrives during another unit time.[1]

In the context of publish/subscribe system, even many index techniques such as multiple one-dimensional indexes based [1], [2], [3], [4], [5] and multidimensional index based [6], [7], have been proposed for event filtering, all these techniques are designed to filter events one by one. They can not deal directly with multiple 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 [6], [7] is more efficient and flexible than those based on multiple one-dimensional indexes [1], [2], [3], [4], [5].

In order to improve average response time of event filtering in the case that multiple events arriving at the same time, in this paper, we first propose an extended R-tree based event filtering algorithm to improve average response time while filtering multiple events arriving at same time. Further, because the number of events arriving at the same time and index data 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 of this paper. Section 3 introduces the algorithm to improve average response time. Section 4 proposes the adaptive model. In Section 5, the algorithm and the adaptive model are evaluated and analyzed in a simulated environment. Section 6 discusses the related work. Finally, conclusion is made in Section 7.

2. Background and Motivation

As introduced in [6], [7], multidimensional index (R-tree or UB-tree) based event filtering is feasible[2] and is much more efficient and flexible than the Count algorithm [3], [4] which is one popular multiple one-dimensional indexes for event filtering. Meanwhile, 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

Even UB-tree and R-tree have different partition strategies, except point query, 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 multiple events arrive at same time.

Because UB-tree partitions space with space filling curve, original UB-tree's search algorithm is depth-first. It's not easy to calculate the number of multiple search paths at one specified middle levels without accessing leaf nodes based on original UB-tree structure. 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. We assume that reader has enough knowledge about R-tree.

3. Improve Average Response Time

3.1 Basic Idea

The basic idea is that for multiple events arriving at same time, their relative workloads for event filtering are

♠ Regular Member Institute of Industrial Science, the University of Tokyo
{botaow, kitsure}@tkl.iis.u-tokyo.ac.jp
♦ Student Member Ph.D student of Graduate School of Information Science and Technology, the University of Tokyo
zhangw@tkl.iis.u-tokyo.ac.jp

[1] Even logically for most of the events, there exist absolutely different arrival times, in this paper, we regards the events arriving in the same unit time as the events arriving at same time.

[2] For details, please refer to [6], [7].

estimated respectively. The workload is defined as the number of search paths at one specified level.

There are two steps for the multiple events filtering algorithm (called *BatchSearch* later): step 1) first, calculate workloads by searching R-tree from root to one specified middle level, step 2) and then do event filtering (R-tree point enclosed query from the specified level) one by one with SJF policy.

Compared to original R-tree, *BatchSearch* has multiple inputs (events) and a new added parameter (called *Level* later) which controls the depth to estimate workloads.

Because the algorithm used at step 2 is similar to original R-tree search algorithm, we introduce only the data structure and algorithm to estimate the workload at step 1.

3.2 Data Structures and Algorithms to Estimate Workload

WorkloadTable is an array of items with structure shown in the left of Fig.1. Each item corresponds to one event. The workload is the number of nodes located at the ending of search paths stopped at the specified *Level*. List of nodes are the corresponding nodes.

IntersectBuffer (right of Fig.1) is used to record events whose MBRs intersect with those of the items in one R-tree node located on the paths at the specified *Level*. The items in one intersect buffer corresponds those of one R-tree node. Each level uses one intersectBuffer while estimating workloads.

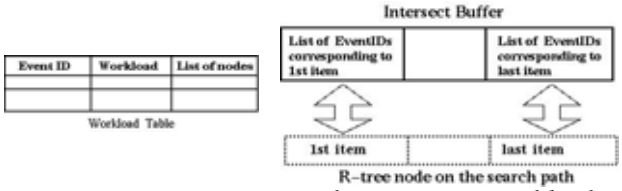


Fig.1 Data Structures Used to Estimate Workload

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

Fig.2 Algorithms to Estimate Workload

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

In function BatchIntersect, line 1-2 read parent item of current node from IntersectBuffer of last level (the level nearer to root) and gets all event IDs kept in the item. Line 3 checks the ending condition of recursive search and line 4 fills the WorkloadTable with the event IDs gotten at line 1-2 and *CurrentNode*. Line 6-15 fill Intersectbuffer of *CurrentLevel*. Line 16-18 search next level by accessing children nodes of *CurrentNode*.

4. Model of Adaptive Search

For same multiple events, the performance is different for different *Level*s which control the depths to estimate workloads. At the same time, the number of multiple events arriving at same time is not fixed, the size and data distribution of index changes dynamically also. In this section, we will propose a self-adaptive model to filter multiple events with average response time same as or close to the possible best time.

4.1 Performance Analysis

While filtering multiple events arriving at same time, time cost to estimate workloads is overhead compared to the processing without workload estimation. The overhead becomes larger with the value increment of the new added parameter *Level*. At the same time, because the higher the *Level* is, the more accurate of workload estimation is, so the efficiency of SJF become more and more better with the value increment of the *Level*. For the same multiple events, the average response time based on the *BatchSearch* is a function of the *Level*. The relationship can be described in the left of Fig.3. Because it has shape of concave logically as shown with mark "**Ideal and logical**", generally the best *Level* exits for the multiple events with same event number. The best means the shortest average response time. The best *Level* changes for different numbers of input events and data distributions of index.

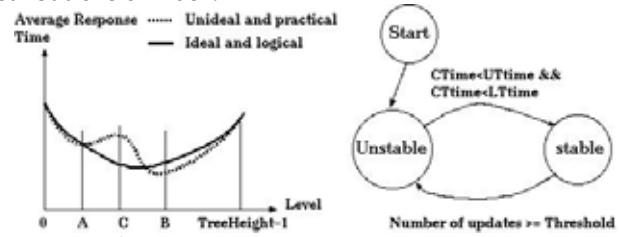


Fig.3 Performance Analysis and Adaptive Model

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

4.2 Adjust Best Level Dynamically According to Statistic Information

The adaptive model is shown on the right of Fig.3, it is built for filtering multiple events with same event number arriving at same time. For multiple events with different sizes (numbers), their statuses will be kept in different

buffers, for example, entries of a status buffer array corresponding to different sizes. The main function of the model is to adjust best level dynamically for filtering multiple events on changing index.

If the current *Level* is best, 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 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.3 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 a dynamic system in practice. The overhead is not neglected for a higher index tree or multiple events with larger number. Our sulution is that check the current Level and its upper and lower levels (totally 3 levels) based on the "**Ideal and logical**" line in Fig.3.

In unstable status, for multiple events (kept in array called *EventArray*) with same size arriving at different time, *BatchSearch* processes these *EventArray*s with *Level* value changed in a loop of round-robin way.

The inputs of *BatchSearch*, multiple events and *Level*, 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 value.

The average response times of three different levels (called *CTime, UTime, LTime* in Fig.3 which correspond to the average response time of current level, upper level, lower level) are summed up and checked after the loop ends. If *CTime < UTime && CTime < Dtime* is true, the system will enter stable status, because the current level is the best level. 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 the content of *EventArray* 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, shown as the line of "**Unideal and practical**" in Fig.3, 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 practica**l" line will change in the same concave shape or close up to " **Ideal and logical**" line statistically as shown in Fig.4-c. The adaptive model is expected to work well if the loop number is big enough. It's manageable for a long time running pub/sub system.

5. Results of Evaluation

5.1 Environment

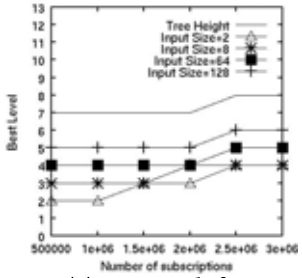
The algorithm is designed for main memory structure and evaluated in a 12D space. Both subscriptions and events are created randomly with unsigned short. The algorithm is implemented based on R*-tree[3] with index node capacity 10 and leaf node capacity 20 which are default values. The event filtering here is point enclosed query. The hardware platform is Sun Fire 4800 with 4900MHz CPUs and 16G memory. The OS is Solaris 8.

5.2 Evaluation

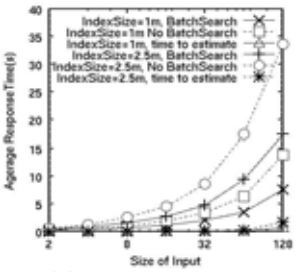
The evaluation results are shown in Fig.4. Fig.4-a shows that the best *Level* changes slowly with increment of index size. The input size means the number of multiple events here.

Fig.4-b compares the average response time of the *BatchSearch* algorithm with best level to that which just inputs events to original R*-tree in a random sequence without considering about workloads. Further, it shows that the cost to estimate workload (algorithm shown in Fig.2 can be neglected compared to average response time. It also shows that the larger the number of event is, the more the average response time can be improved. The maximum is nearly up to 50% in our evaluation.

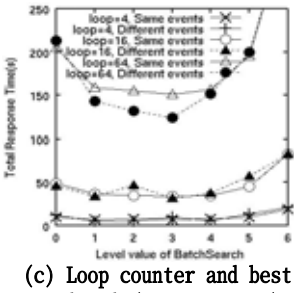
Fig.4-c shows an example of the changing trend of average response times which are calculated with same and different events gotten at different levels. There the index size is 1.5 million and the height of tree is 7。It shows that with increment of loop counter, the trembling of response time marked "**Unideal and practical**" in Fig.3 and captured in Fig.4-c (loop=4 and loop=16, Different events), becomes more and more stable with shape of concave and merge into the line gotten with the same events at each level.



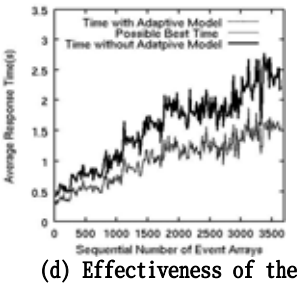
(a)Best Level for BatchSearch



(b)Effectiveness of BatchSearch



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



(d) Effectiveness of the adaptive Model

Fig.4 Evaluation of BatchSearch and Adaptive Model

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

The Fig.4-d shows the performance of unstable status. It is compared to the performance without using the adaptive model 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 objects (subscriptions) are inserted into the index. So Fig.4-d shows 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.4-b which are gotten at stable status. Even in unstable status, the performance of the adaptive model is very close (too close to make difference here) to the possible best performance as shown in Fig.4-d. We can say that with the adaptive model, multiple events can be filtered with response time close to the possible best time. The difference can be neglected compared to the performance without the adaptive model.

6. Related Work

A lot of algorithms related to event filtering have been proposed. They are proposed for publish/subscribe systems [2], [4], [5], [6], for continuous queries [1], [8], [9] and for active database [3].

Predicate indexing techniques have been widely applied. There, a set of one-dimensional index structures are used to index the predicates in the subscriptions, the representative algorithm is called counting algorithm [4] and Hanson algorithm [2], [3]. They differ from each other by whether or not all predicates in subscriptions are placed in the index structures.

In [6], [7], multidimensional index structures based event filtering is proved to be feasible and efficient. It's the basis of this paper.

Event filtering is one critical step of continuous queries. In [1], predicate index is built based on Red-Black tree, there algorithm is similar to bruteforce search that scans the total Red-Black tree for event filtering each time. Count algorithm is used in [8], [9]. One routing policies is implemented in [9] to let faster operators filter out some tuples before they reach the slower operators. In [10], 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

In this paper, we first proposed an event filtering algorithm with multiple events as input based on an extended R-tree search algorithm for publish/subscribe system. Further an adaptive model was designed to filter multiple events for different event numbers and changing index. According to the evaluation results, the average response time can be improved maximally 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.

[References]

[1]S. Chandrasekaran and M. J. Franklin: “Streaming queries over streaming data”, VLDB, pp.203-214(2002)

[2] 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”, SIGMOD, pp.115-126(2001)

[3]E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. B. Park, and A. Vernon: “Scalable trigger processing”, ICDE, pp.266-275(1999)

[4]T.W.Yan and H.Garcia-Molina: “The shift information dissemination system”. ACM Trans. Database Syst., 24(4), pp.529-565(1999)

[5] B.Wang, W. Zhang, and M. Kitsuregawa: “Design of B+tree-based predicate index for efficient event matching”, APWEB, pp.537-547(2003)

[6] B.Wang, W. Zhang, and M. Kitsuregawa: ” UB-Tree based efficient predicate index with dimension transform for pub/sub system”, DASFAA, pp63-74.(2004)

[7]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

[8]J. Chen, D. J. DeWitt, F. Tian, and Y.Wang: ” Niagaracq: a scalable continuous query system for internet databases”, SIGMOD, pp. 379-390(2000)

[9] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman: ”Continuously adaptive continuous queries over streams”, SIGMOD, pp.49-60 (2002)

[10] S. Viglas and J. F. Naughton: “ Rate-based query optimization for streaming information sources”, SIGMOD, pp.37-48(2002)

Botao WANG

Research Fellow at Institute of Industrial Science, the University of Tokyo. He received the Ph.D degree in computer science in 2000 from Kyushu University. His research interests include spatial-temporal database, parallel processing and data stream. He is a regular member of DBSJ.

Wang ZHANG

Ph.D courses student of Graduate School of Information Science and Technology, the University of Tokyo. He received the Master degree in information engineering in the current gradate school. His research interests include data clustering and data stream. He is a student member of DBSJ.

Masaru KITSUREGAWA

Professor and the director of center for information fusion at Institute of Industrial Science, the University of Tokyo. He received the Ph.D degree in information engineering in 1983 from the University of Tokyo. His research interests include parallel processing and database engineering. He is a member of steering committee of IEEE ICDE, PAKDD and WAIM, and had been a trustee of the VLDB Endowment. Now he servers as general co-chair of ICDE2005. He was the chair of data engineering special interest group of Institute of Electronic, Information, Communication, Engineering (IEICE), Japan and the chair of ACM SIGMOD Japan, Chapter. He is currently a trustee of DBSJ.