DEWS2006 2A-o4

Efficient Large Scale Continuous Selection-Join Queries Based on Multidimensional Index

Botao WANG† and Masaru KITSUREGAWA†

† Institute of Industrial Science, The Univeristy of Tokyo
Komaba 4–6–1, Meguro-Ku, Tokyo, 153–8505 Japan

Abstract We consider the problem of large number of continuous selection-join queries over data streams. As far as we know, in current data stream management systems, events are filtered based on the query plan(s) which are created according to continuous queries defined by users. Even many kinds of optimizations on query plans have been proposed, there are few proposals on the processing of continuous selection-join queries. The query plan-based processing of continuous selection-join queries schedules the executions of operators and manages the buffers for intermediate results used by selection or join operators. When the number of continuous selection-join queries becomes larger, the processing based on query plan is inefficient for two reasons: 1)the system has to schedule large amount of operators, and 2) the memory will be consumed quickly by the larger number of buffers used by the operators. In this paper, we introduce an efficient event filtering algorithm based on multidimensional index structure, where the event filtering is considered as a query in multidimensional space. Because our proposal is not based on query plan, no operator scheduling is needed. At the same time, only one global buffer for join operation is kept, accordingly much larger sliding window can be defined. We first introduce an event filtering model for continuous selection-join queries. Based on the model, an optimized event filtering algorithm is proposed to filter the stream of join results efficiently. The evaluation results show that the event filtering of large number of continuous selection-join queries based on our model has good scalabilities with respect to the number of the queries and number of dimensions. The performance of the event filtering is improved significantly by the proposed algorithm, especially for the continuous selection-join queries with large sliding window.

Key words data stream, multidimensional index, performance evaluation

1. Introduction

Many kinds of stream data are being generated from different sources, like stock market, weather forecast, location information (cars, kids, pets), logs (web access, telecommunication, click), sensors (buildings , animals, patients), etc.. Continuous Queries (CQ) are defined to monitor and process those data in data stream management systems. Some of those data from same or different sources are relevant. For example, for a future trader, he may concern on both forecast of a place where one commodity being traded by him is produced and the current price of the commodity. One example CQ corresponding to his request is shown as follows.

SELECT $*$
FROM $TradingStream\ T$, $WeatherStream\ W$
WHERE $T.name =$ "*orange*" AND $T.price < 25$ AND
$T.productionArea = W.area$ AND
$W.event =$ "*hurricane*" AND $W.category > 4$ AND
$W.date < August$

The number of such kind of CQs may be hundreds of thousands. In the paper, we consider the problem of large number of continuous selection-join queries over data streams for this kind of data stream applications. The Continuous Selection-Join Query (CSJQ) is the continuous query composed of both join and selection operations as shown in the example.

As far as we know, up until now, there doesn't exist an efficient data stream management system which can deal with larger number of CSJQs on multiple streams. Even the systems like Telegraph [7] [8] [17], Niagara [10] [9], STREAM [2] [3] [4], Aurora [1] [20], Gigascope [16], can process CSJQs, we argue that it is hard for these systems to process CSJQs when the numbers of CSJQ and stream become larger. The main reason is that the event filtering of CSJQs in [1] [2] [3] [4] [9] [10] [16] are based on a global query (execution) plan composed of pipelined operators, which have their own buffers. The overhead to schedule large number of operators can not be neglected. The maintenance of the buffers used by the operators is complicated. The more the number of buffers is, the more the memory will be used. [8] has problems of space or maintenance when the numbers of CSJQs and tuples kept in buffer become large.

In order to overcome the above problems, we propose an efficient algorithm for large number of CSJQs on multiple streams. The main contributions of this paper are

- We propose a simple event filtering model for CSJQs. The model is based on multidimensional index structure. No pipelined operator is used. Only one global buffer is used to keep tuples for join operation. It allows users to define a large number of CSJQs with large sliding window on multiple streams.
- We propose an algorithm to filter join results efficiently. The performance can be improved significantly compared to the naive algorithm.

The rest of the paper is organized as follows. Section 2 presents the event filtering model for CSJQs. Section 3 describes the join result filering algorithm. Section 4 presents the experimental results. Section 5 discusses the related work. Section 6 contains the concluding remarks and future work.

2. Event Filtering Model for Continuous Selection-Join Queries

We first analyze the reasons why current query plan-based solutions can not deal with large number of CSJQs and then introduce the motivation of our proposal.

2.1 Preliminaries

There are two kinds of basic operations in CSJQ queries: selection and join. Selection is a stateless operation that no buffer is required to keep past data. Join operation needs buffers to keep past data. While creating a global query plan for the total CSJQs, the system has two basic strategies (Niagra [9] [10]) to arrange the execution sequence of selection and join operators: selection first or join first as shown in Fig.1.

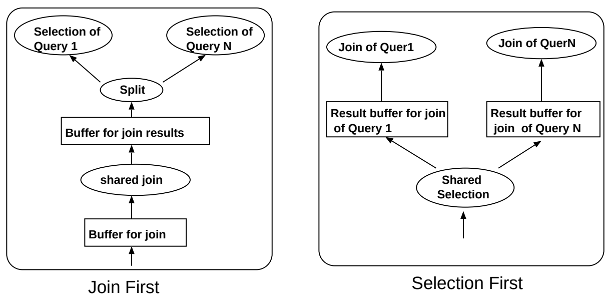

Figure 1 Two strategies of creating query plan for multiple CSJQs

For the join-first query plan, only one buffer is used to keep the results of the shared join operation. The problem is that when the number of CSJQs is larger, the number of different selection operators becomes larger also. The overhead to schedule the executions of these operators is high. For the selection-first query plan, besides the scheduling problem same as join first, it requires large number of buffers for join operations. More memory spaces are used. Without sharing operations at the first step (multiple query plans), the system will suffer in the overhead of scheduling and high memory cost much more.

As introduced in [9], join-first plan is better than selection-first plan for the reason that join is a more expensive operation than selection. The selection operation filters events according to predefined predicates. Many predicate index techniques for event filtering have been proposed, which are built based on multiple one-dimensional index structures (Count algorithm [25] and Hanson algorithm [11] [14] [15]) or multidimensional data structure [21] [22]. As introduced in [21] [22], multidimensional data structure is feasible for event filtering by utilizing efficient point access methods like UB-tree [19] or MultiLevel Grid File (MLGF) [23] [24].

Our motivation is that filtering stream of join results (output by "shared join" in Join First, Fig.2) by utilizing predicate index techniques introduced above instead of multiple selection operations, so the overhead of scheduling multiple selection operators can be avoided.

2.2 Event Filtering Model for CSJQs

The model is shown in Fig.2. It is the model for all CSJQs with same join predicate. For example, all the CSJQs with join predicate $T.productionArea = W.area$ will be processed in the same model.

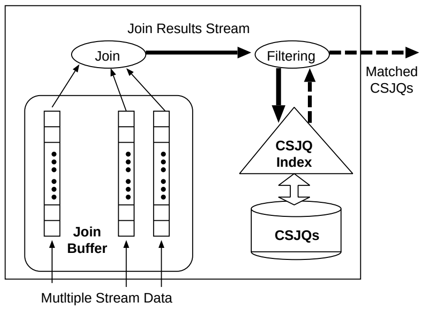

Figure 2 Event filtering model for continuous selection-join queries

Inside the model, there are 2 data structures (join Buffer and CSJQ index) and 2 corresponding operations (Join and Filtering). Join Buffer is used to keep past data for join operation. It is a global buffer shared by all the CSJQs with same join predicate. CSJQ index is a predicate index to keep the CSJQs.

The event filering of CSJQs has two steps. The first step is join operation where stream data are joined. Symmetric hash join is used in our model. The result is a stream. The second step is filtering join result stream by utilizing the CSJQ index.

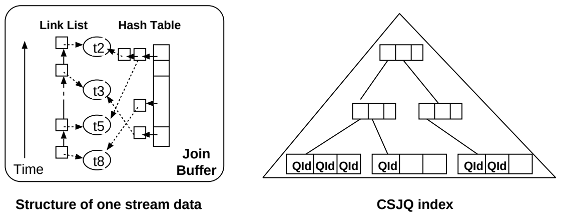

Figure 3 Data structure used in event filtering model of continuous selection-join queries

Fig.3 shows the details of the model. Each stream has two data structure: link list and hash table. The link list is used to support the insertion and deletion of stream data in the way of first in first out. Hash table is used for hash join. The data type of both link list and hash table is pointer of the tuple belong to the stream. For each tuple, there is only one copy in the join buffer.

The CSJQ index is a multidimensional index with tree structure. Its input is a join result and output is a set of matched CSJQs. Each dimension corresponds to one attribute of stream and all attributes used to define selection predicates of CSJQ are used. As an illustration, the example query used in Section 1, for two streams: 1) TradingStream with attributes (*name*, *price*, *productionArea*) and 2) WeatherStream with attributes (*event*, *aera*, *category*, *date*), the attributes used in CSJQ index are (*name*, *price*, *productionArea*, *event*, *aera*, *category*, *date*). The filtering of one join result can be a 7d point-enclosed query on hypercubes in a 7d space or a 14d range query on points in a 14d space by dimension transform. For the details of the way to make use of multidimensional data structure, please refer to [21] [22]. The same way is applied to build CSJQ index for multiple streams.

Because there is no operator inside the model, the overhead related to execute multiple operators individually can be avoided. Even each kind of join predicate corresponds to one instance of the model. We believe that the number of different join predicates is much less than the number of CSJQs. At the same time, for one model, only one global buffer is used to keep past data for join operation, so the buffer can be very large.

3. Optimization of Join Result Filtering

In the next, we first introduce some assumptions in this paper and then introduce an optimized algorithm to filter join results in the case that the number of join results is more than one.

3.1 Assumptions

Due to the unbounded nature of streams, queries over streams are often defined in terms of sliding window. In this paper, we address sliding windows that are applied across all streams and the window is defined in terms of period of time. We only consider about sliding-window equijoin in this paper in order to simplify the problem.

3.2 Observations on Join Results and Search Paths

In our event filtering model for CSJQs, a link list and a hash table are used for each stream, the cost related to join operation is low because symmetric hash join is used. The cost of filtering join results dominates the total cost of CSJQ processing. Especially the number of join results is larger.

The naive method to filter join results is to make use of the original search algorithm of the multidimensional data structure selected to build CSJQ index. Here we propose an optimized algorithm to improve the performance of join result filtering based on the following two observations:

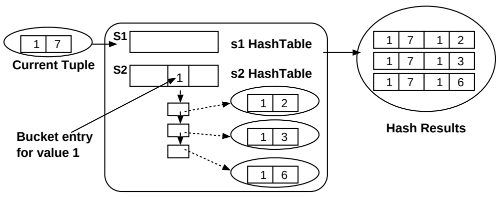

Figure 4 Structure of hash join result (Input of CSJQ index)

- There exists a common part among the join results (input of CSJQ index). Fig.4 shows an example, where there are two streams S1 and S2. The attribute used by join operation of each stream is the first attribute here and the current input is a tuple belongs to S1. As shown on the right of Fig.4, the common part is the set of data of the attributes belong to S1 and the value of each attribute has the same value as the corresponding value of the S1 input tuple ([1,7]).
- The search result set of a multidimensional index can be regarded as an intersection of two result sets obtained from two excluded dimension (attribute) sets of which the total dimension space is consisted as shown on the top of Fig.5. There, one dimension set consists of all attributes belong to stream S1 and another set consists of all attributes belong to stream S2.

Generally there are several strategies (depth-first, right-first or leaf-first) for the traverse of index tree. For all these strategies, the current input tuple (S1:[1, 7] in Fig.4) will be processed multiple times in the case that the number of join result is more than 1. It is a kind of overhead. In the next, an optimizing algorithm to eliminate this overhead is

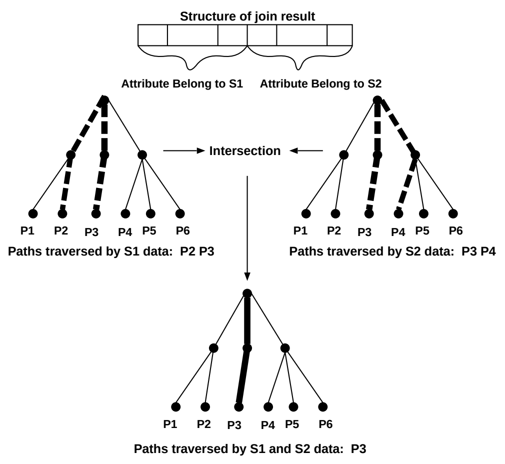
Structure of join result
Attribute Belong to S1
Attribute Belong to S2
Intersection
P1 P2 P3 P4 P5 P6
Paths traversed by S1 data: P2 P3
Paths traversed by S2 data: P3 P4
Paths traversed by S1 and S2 data: P3

Figure 5 Paths traversed by different dimension sets

introduced.

3. 3 Partial Dimension Filtering Algorithm (PDFA)

The optimization is based on a search algorithm called Partial Dimension Filtering Algorithm (PDFA). The basic idea of PDFA is that only the dimensions specified by current input(stream) will be checked while traversing index tree. If PDFA is executed multiple times with different dimension sets and the input of each execution is the output of last execution, the final result set will be same as that of the original search algorithm. Fig.6 shows an example where the PDFA is executed 2 times. The first execution (Step1) with dimension set belongs to S1 (Fig.4), is executed on the total index tree, and the second execution (Step 2) is on a subtree of the index tree composed of the paths leading to the results of the first execution. The dimension set of the second execution is the set of dimensions belong to S2 (Fig.4). When the number of join results is more than 1 as shown in Fig.4, the first step needs to be executed only one time. It is the point of our optimization.

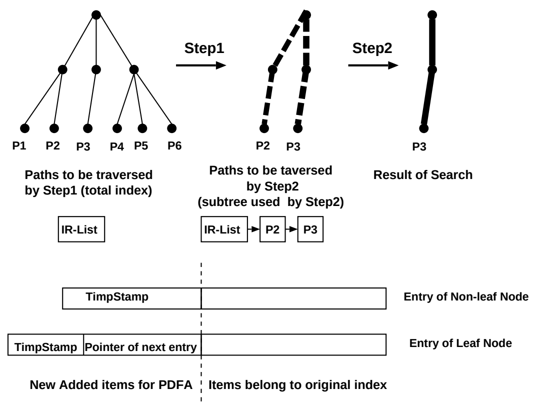

Figure 6 Search based on PDFA and the related data structure

In order to filter join result by PDFA, the paths leading to the results of the previous execution must be remembered. The data structures of the entries inside non-leaf node and leaf node of the index tree are extended as shown in Fig.6. "TimeStamp" is a flag which is used to mark the paths (on which the results were found) of the first execution and is used as an indicator for the second execution. For the entry belongs to leaf node, an pointer named "Pointer of next entry" is defined, which is used to point to the next entry which is the result of the first execution. As shown in Fig.6, P3 is pointed by P2 after the first execution. The system will keep the head of this list named Intermediate Result (IR) list. The reason we define the pointer is that the second execution does not need to be executed from root to leaf when the tree is high and the number of intermediate result is small. It is an implementation problem.

In order to execute PDFA on the index tree, some basic functions used by multidimensional data structure (intersect, include, inside, etc.) must be redesigned. The pseudo code of inside function (named **insideEntry**) used by PDFA is shown in Fig.7. The output of the function **insideEntry** is a boolean value. The input of *currentEntry* and *Range* have same meanings as those of the original inside checking function. The *flag* is the sequence No.(FIRST or SECOND) of the current execution. The *currentStamp* is the system time just before the first execution and the same value will be used as the input of the second execution. *dimSet* is the set of dimensions will be used in current execution. Line 4-6 check whether the entry has been marked or not if the current execution is not FIRST. If the entry was not marked, the entry needs not to be processed further. Line 8-12 check whether the current entry is inside the input *Range* with regard to the dimensions in *dimSet*. Line 14 means, if the *currentEntry* values of all dimensions is inside the corresponding *range*, the *timeStamp* (Fig.6) of *currentEntry* is set to be *currentStamp* in the case that current execution is FIRST. The redesigns of the other basic functions are similar to the above example. The maintenance of the IR-list is straight-forward.

```
1  bool
2  insideEntry(currentEntry, Range, flag, currentStamp, dimSet)
3  Begin
4      If (flag != FIRST && currentStamp != currentEntry.TimeStamp)
5          return FALSE;
6      EndIf
7
8      For (For each dimension currentDim in DimSet)
9          If (Range on currentDim is not inside currentEntry)
10             return FALSE;
11         Endif
12     EndFor
13
14     if (flag == FIRST)
15         currentEntry.timeStamp = currentStamp;
16     EndIf
17
18     return TRUE;
19 End
```

Figure 7 Inside function used by PDFA

3. 4 Efficiency of PDFA

Many factors influence the efficiency of PDFA. Fundmentally, the more the number of join result is, the more the filtering based on PDFA can be benefited. Considering the cost of join operation is low here, the average response time of one event filtering of CSJQ can be formulated by the following equation,

$$T_{average} = T_{Step1} + T_{Step2} * N_{join} \tag{1}$$

where $T_{average}$ represents the average response time, T_{Step1} and T_{Step2} corresponds to the time of Step1 and Step2 in Fig.6, N_{join} represents the number of join results. As shown in Fig.6, because Step1 is executed on the total index tree and Step2 is executed on the subtree or IR list directly. Generally, T_{Step2} is less than T_{Step1}. And for the same reason T_{Step2} is less than the average response time based the original search algorithm. The larger the N_{join}, the higher the efficiency of PDDA compared to the original search algorithm is. When T_{Step1} is much larger than T_{Step2} (Fig.11), the $T_{average}$ is dominated by T_{Step1} which is a constant if N_{join} is not large enough to make $T_{Step2} * N_{join}$ to be comparable to T_{Step1}. We think this observation is very meaningful in practice.

Because large silding window means that more join results will be produced, we can say that PDFA is suitable to support large sliding window.

4. Evaluation

4. 1 Environment

We implemented and compared 4 approached based on 2 algorithms with 2 multidimensional index structures Rtree [5] [12] and MultiLevel Grid File (MLGF) [23] [24]. The two algorithms are:

- Naive. The filtering algorithm of CSJQs is same as the original one of the corresponding multidimensional index data structure. The selection predicate of CSJQ is regarded as a hypercube in a high dimensional space and kept in Rtree. While using MLGF, the hypercube (selection predicates) in a d space is transferred into a point in a 2d space. The method of dimension transform and its efficiency has been reported in [22] [21].
- Optimized. The ways (index building and dimension transform) are same as those of naive approaches. The difference is that join results were filtered based on PDFA.

The stream data and continuous selection-join queries are created according to a workload specification as shown in Table 1 along with their default values. we change one parameter and fix the others with their default values without specific introduction. The size of sliding window is defined according to period of time. For each evaluation, all results were measured after all the related data had been loaded into main memory. The sizes of sliding windows of all CSJQs queries are same. The average response time of 1000 tuple filterings of CSJQs was measured for performance evaluation.

Table 1 **Parameters Related to Workload**

Parameter	**Value range**	**Default value**
Number of queries	50,000-500,000	100,000
Number of attributes (dimension) per stream	3-17	5
Number of streams	2-6	2
Selectivity of CQSJ index	0.001%-0.1%	0.01%
Ratio of dimension of two streams	1:5 - 5:1	1:1

All the solutions were implemented in C++. The type of all attributes used by selection predicates is short integer, and the type of the attribute used by hash key is integer. The hardware platform is a Sun Fire 4800 workstation with four 900MHz CPUs and 16G bytes memory under Solaris 8.

4. 2 Evaluation of Partial Dimension Filtering Algorithm

4. 2. 1 Scalability with Respect to Number of Queries

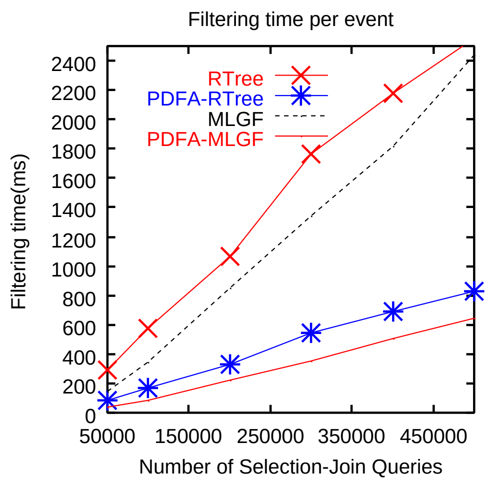

Figure 8 The Scalability with Respect to Number of Queries

Fig.8 shows the scalability on the number of queries. The number of queries changed from 50,000 to 500,000. **MLGF** stands for naive solutions based on MLGF. **PDFA-MLGF** stands for optimized solution based on MLGF using PDFA. **RTree** and **PDFA-RTree** have corresponding meanings. All these 4 legends symbol have same meanings in the followed evaluations.

Fig.8 shows that the MLGF-based implementations outperform the RTree-based implementations. The Implementations based on PDFA outperforms the naive implementations. All solutions have good scalabilities. We can say that the proposed model is suitable to deal with larger number of CSJQs.

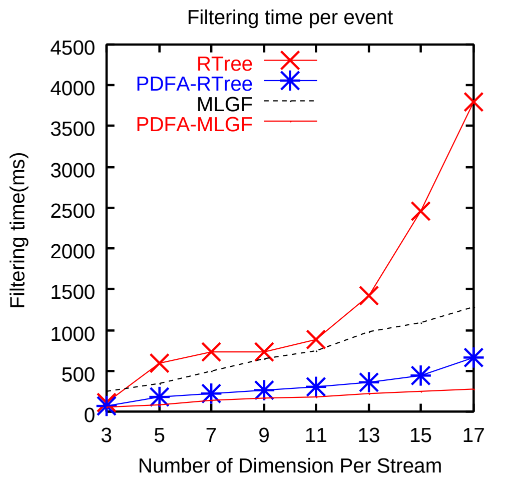

Figure 9 The scalability with respect to dimension

4. 2. 2 Scalability with Respect to Number of Dimensions

Fig.9 shows the scalability with regard to the number of dimensions (attributes) in data streams. X-axis represents the number of dimensions used by one stream. The MLGF-based implementations has good scalability in our environment.

The performance of RTree-based implementation deteriorates quickly when the number of dimensions increase. The reason is that one CSJQ is used as a hypercube in a d space for RTree-based implementation. The overlaps between hypercubes increases exponentially when the number of dimensions increases. The MLGF-based implementation is a range query on points in a $2d$ space. No overlap exists between points.

4. 2. 3 Scalability with Respect to Selectivity of CSJQ Index

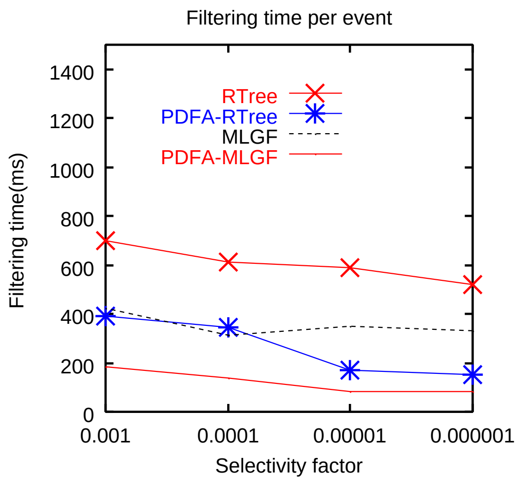

Figure 10 The scalability with respect to selectivity of CSJQ index

Fig.10 shows the scalability with regard to the selectivity of CSJQ index. Except the naive implementation based on MLGF, the average response time decreases with selectivity. We found that the numbers of leaf nodes accessed by MLGF with different selectivities are at same level, even the number of results decreases while the selectivity becomes less, the related time doesn't change much. The reason is that the hypercube representing the selection predicate in a d space is transferred into a point in a $2d$ space. The data distribution are total different from the original distribution. It is one problem of event filtering based on multidimensional index structure with dimension transform introduced in [21] [22].

4. 2. 4 Scalability with Respect to Number of Join Results

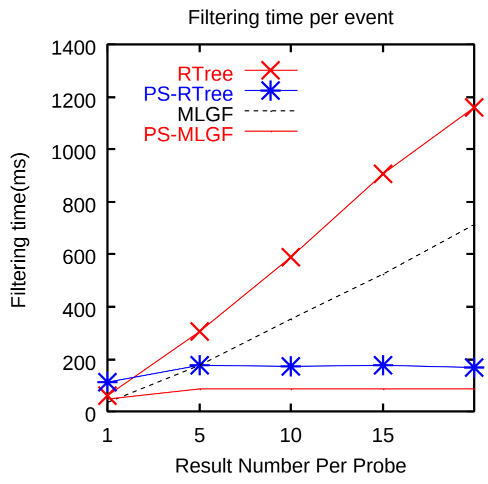

Figure 11 The scalability with respect to number of hash result

Fig.11 shows the scalability with regard to number of hash join results. The number of hash result was controlled by adjusting the number of tuples kept by each stream. It is reasonable that the average response time increases when the number of join results becomes larger for the naive implementations. Because its performance is linear to the number of input. The average response time based on PDFA seems to be a constant when the number of join results is larger than 5 in our environment. The reason is that T_{Step1} is nearly 4 orders of magnitude larger than T_{Step2}. According to equation (1) in Section 3.4, the performance mainly depends on T_{Step1}. The time related to $T_{Step2} * N_{join}$ can be neglected. The result shows that PDFA can support larger sliding window efficiently, because the larger the sliding window is, the larger the number of tuples pointed by hash tables is.

Next evaluation shows the performance when the number of join results increases in a different way.

4. 2. 5 Scalability with Respect to Number of Streams

Fig.12 shows the scalability with regard to number of streams. With the increasement of number of streams, the number of join results increases exponentially if the average number of one probe is larger than 1. Here the number of per hash probe is set be 2. So the number of join results changes

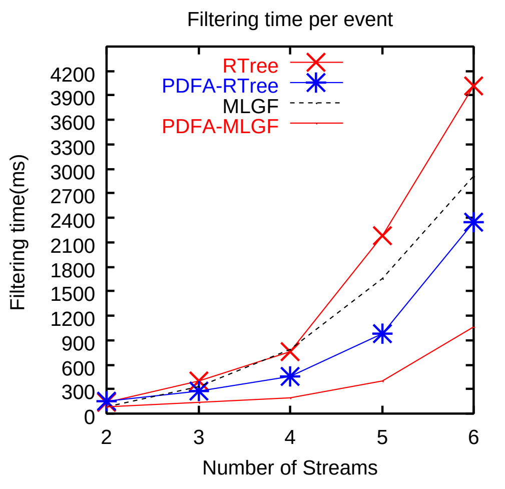

Figure 12 The scalability with respect to number of streams

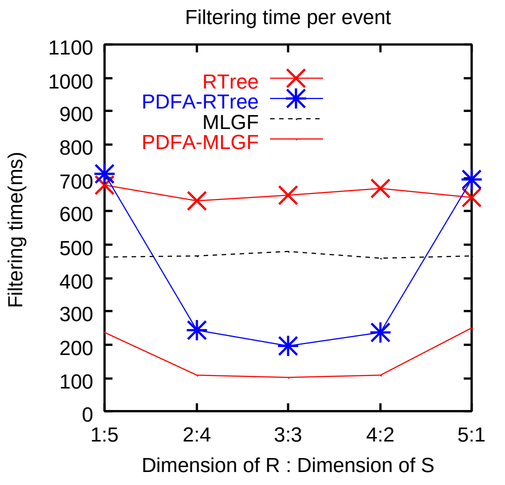

Figure 13 The scalability with respect to dimension distribution via two streams

in the sequence 4, 8, 16, 32, 64 when the number of streams increases from 2 to 6. Even the performance based on PDFA outperforms that based on naive, the improvement is not as large as that in the evaluation introduced in Section 4.2.4. The difference between two evaluations is the number of dimensions while filtering join result. In evaluation in Section 4.2.4, Step1 and Step2 (Fig.5) have same number of dimension. In current evaluation Step1 and Step2 have different numbers of dimension. The difference becomes larger with the increasement of number of streams. Logically this evaluation is similar to the evaluation (Fig.13) to be introduced in next section.

4.2.6 Scalability with Respect to Ratio of Dimensions of Two Streams

Fig.13 shows the scalability with regard to respect to ratio of dimensions of two streams. The total dimension number of 2 streams is 12 dimensions, but the ratio of dimensions is different.

Fig.13 shows that when the dimensions of two streams is same, the improvement is best. The reason is same as that introduced in Section 4.2.4. At the same time, when the ratio of dimensions between two streams become larger, the performance improvement based on PDFA become less also. The reasons are that 1) when the number of dimensions of two streams is equal, T_{Step1} is much larger than T_{Step2} and T_{Step2} can be neglected with regard to the total cost, 2)when the number of dimensions of two stream is different, the larger the difference is, the larger the T_{Step2} is. T_{Step2} can not be neglected any more. That's reason why the performance improvement based on PDFA decrease when the number of streams increases in Fig.12.

5. Related Work

Many prototype systems which support continuous selection-join have been proposed. In STREAM [18], queries are independent units that logically generate separate plans. Aurora [6] uses one global query plan performing all computation of interest to all users, who build queries out of a small set of operators. It uses a single globally-shared queue for inter-operator data flow instead of separate queues between operators. GigaScope [16]is a stream-only database, it doesn't not support stored relations or continuous queries. [13] shares the execution of the different window joins to optimize the utilization of system resources. There the shared join produces multiple output data streams for each separate query. For Telegraph [8] [17], the results of all predicates correspond to all tuples are kept in a buffer in [8]. When the number of queries and the size of sliding window is larger, the size of the buffer is large for the reason that the number of predicates increases. In [17], for each query, a set of status bits are defined and inserted on each tuple which is routed among kinds of operators. The size of tuple routed among the operators will become larger when the number of queries is larger.

For theses systems, the execution of query operators is controlled by a global scheduler except [8]. Even resources and computation can be shared with query plans, when the number of queries or the size of sliding window become large, it becomes difficult for system to schedule hundred thousands operators and the buffers used by theses operators need large amount of memory resources also.

Our proposal is not based on query plan(s), so it doesn't have scheduling problem and its realted memory problem. The performance mainly depends on the multidimensional data structure chosen to filter join results. According to our evaluation, MLGF is one candidate of the feasible multidimensional data structures. We evaluated our proposal on multiple streams.

6. Conclusions and Future Work

Based on multidimensional data structure, we introduced an event filtering model for large number of continuous selection-join queries on multiple streams. No operator scheduling is needed and only one global join buffer is used. We proposed an optimized algorithm called partial dimension filter algorithm to share the computations while filtering join results.

In an experimental study with synthetic data, our experimental results showed significant improvement in event filering time. The event filtering of continuous selection-join queries based on our proposal has good scalabilities with regard to the number of queries and number of dimensions.

We intend to pursue the following 2 directions in future. 1)Besides continuous selection-join queries, there are many other kinds of continuous queries including aggregate operation, sort operation, group by operation, etc.. We want to extend our model to support continuous queries including the other types of operations besides selection and join operations. 2)The PDFA is sensitive to the ratio of dimensions used by different streams, we want to investigate whether the PDFA can be more efficient by filtering multiple streams with multiple steps, because join of multiple streams requires multiple probes, which produce subsets of join results with sharable computations.

References

[1] D. J. Abadi, D. Carney, U. etintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[2] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 407–418. ACM Press, 2004.

[3] S. Babu, K. Munagala, J. Widom, and R. Motwani. Adaptive caching for continuous queries. In *ICDE*, pages 118–129, 2005.

[4] S. Babu, U. Srivastava, and J. Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Trans. Database Syst.*, 29(3):545–580, 2004.

[5] 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.

[6] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Monitoring streams - a new class of data management applications. In *VLDB*, pages 215–226, 2002.

[7] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[8] S. Chandrasekaran and M. J. Franklin:. Streaming queries over streaming data. In *VLDB*, pages 203–214, 2001.

[9] J. Chen, D. J. DeWitt, and J. F. Naughton. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *ICDE*, pages 345–356, 2002.

[10] 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 Man agement of data*, pages 379–390. ACM Press, 2000.

[11] F. Fabret, H. A. Jacobsen, F. L. bat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscrib e systems. In *SIGMOD*, pages 115–126, 2001.

[12] A. Guttman. R-trees: a dynamic index structure for spatial searching. *Readings in database systems*, pages 599–609, 1988.

[13] M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid. Scheduling for shared window joins over data streams. In *VLDB*, pages 297–308, 2003.

[14] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. B. Park, and A. Vernon. Scalable trigger processing. In *ICDE*, pages 266–275, 1999.

[15] E. N. Hanson, M. Chaabouni, C.-H. Kim, and Y.-W. Wang. A predicate matching algorithm for database rule systems. In *SIGMOD*, pages 271–280, 1990.

[16] T. Johnson, S. Muthukrishnan, V. Shkapenyuk, and O. Spatscheck. A heartbeat mechanism and its application in gigascope. In *VLDB*, pages 1079–1088, 2005.

[17] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD Conference*, pages 49–60, 2002.

[18] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. S. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

[19] F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, and R. Bayer. Integrating the ub-tree into a database system kernel. In *VLDB*, pages 263–272, 2000.

[20] N. Tatbul, U. Çetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *VLDB*, pages 309–320, 2003.

[21] B. Wang and M. Kitsuregawa. Dimension transform based efficient event filtering for symmetric publish/subscribe system. In *DEXA*, pages 786–796, 2005.

[22] B. Wang, W. Zhang, and M. Kitsuregawa. UB-Tree based efficient predicate index with dimension transform for pub/sub system. In *DASFAA*, pages 63–37, 2004.

[23] K. Y. Whang, S. W. Kim, and G. Wiederhold. Dynamic maintenance of data distribution for selectivity estimation. *The VLDB Journal*, 3(1):29–51, 1994.

[24] K.-Y. Whang and R. Krishnamurthy. The multilevel grid file - a dynamic hierarchical multidimensional fil e structure. In *Proceedings of the Second International Symposium on Database Syst ems for Advanced Applications*, pages 449–459, 1992.

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