BMQ-Index: Shared and Incremental Processing of Border Monitoring Queries over Data Streams

Jinwon Lee, Youngki Lee, Seungwoo Kang, SangJeong Lee, Hyunju Jin, Byoungjip Kim, Junehwa Song
Korea Advanced Institute of Science and Technology
{jcircle, youngki, swkang, peterlee, hyunju, bjkim, junesong}@nclab.kaist.ac.kr

Abstract

Border Monitoring Query (BMQ) has different query semantic from conventional continuous range query. It monitors the values of data streams and reports them only when data streams cross the borders of its range. In this paper, we first emphasize the importance and usefulness of BMQ through attractive service scenarios. Then, we propose BMQ-Index, which is specialized to BMQ evaluation. It efficiently processes a large number of BMQs in a shared and incremental manner. For shared processing, BMQ-Index adopts a query indexing approach, thereby achieving a high level of scalability. For incremental processing, BMQ-Index employs an incremental access method. Thus, successive BMQ evaluations are significantly accelerated. We present an index structure and a search algorithm to support one-dimensional as well as multi-dimensional BMQ. Lastly, we demonstrate the performance benefits of BMQ-Index through analysis and experiments.

1. Introduction

Advances in mobile computing and embedded device technologies open up new computing environments. The environments contain numerous data generators such as sensors, probes and agents, which generate data in the form of continuous data stream. In order to monitor such data streams and take proper actions, if needed, users register a large number of range queries or filters which are evaluated continuously [1][2][3][11][12][14].

One of primary concerns in the data stream monitoring is to know which data streams begin or end to satisfy range conditions of queries. In many cases, users are interested in knowing whether a data stream falls within a range or not. It is useful enough to report the beginning and end of satisfying range conditions rather than all satisfying events. In addition, the beginning and end information is compelling to users who want proper actions to be automatically triggered or stopped.

In this paper, we first characterize a new type of continuous range query, namely *Border Monitoring Query* (*BMQ*). It only reports data coming into or going out from a query range, i.e., data crossing the borders of a query range. Note that the semantic of BMQ is different from that of a existing continuous range query, namely *Region Monitoring Query* (*RMQ*) [2][12][14][17]. It reports all matching data in a query range. In spite of the usefulness of BMQ semantic, none of previous research on data stream processing [1][3] has developed a special mechanism for BMQ evaluation although many efficient mechanisms are proposed for RMQ evaluation.

To address the challenge, we propose BMQ-Index, an efficient query index specialized to BMQ evaluation. The main idea of BMQ-Index is *shared* and *incremental* processing. For shared processing, BMQ-Index adopts a query indexing approach, thereby achieving a high level of scalability. Once BMQ-Index is built on registered queries, only relevant queries are quickly searched for upon an incoming data. For incremental processing, BMQ-Index employs an incremental access method, i.e., an index structure to store delta query information and an incremental search algorithm. Thus, successive BMQ evaluations are greatly accelerated.

Based on the main idea, we design a one-dimensional BMQ-Index structure and a search algorithm. The one-dimensional index divides the range of possible data values into *Region Segments* by the borders of queries. It stores a query into only two segments where the query range starts and ends. Upon an incoming data, border-crossed queries are incrementally derived during linear traversals from a previous matching segment to a current matching segment. We also design multi-dimensional BMQ-Index by directly extending one-dimensional BMQ-Index. For multi-dimensional search operation, we additionally develop a *cross-check algorithm*.

BMQ-Index has two important features: excellent search performance and low storage cost. As mentioned before, the shared and incremental processing enables BMQ-Index to achieve remarkable search performance. Also, BMQ-Index only needs to maintain delta query information, which consumes a small size of memory space. Such low storage cost is essential in data stream processing where only in-memory algorithm is practical. Compared to the straightforward approach based on state-of-the-art RMQ evaluation mechanism, BMQ-Index achieves much better search performance and storage cost.

The contribution of this paper is summarized as follows. First, we characterize a new type of continuous range query semantic, i.e., Border Monitoring Query, and formally define it. We also show its usefulness with attractive service scenarios. Second, we develop BMQ-

Index which evaluates a large number of BMQs in a shared and incremental manner, thereby achieving excellent search performance and low storage cost. Finally, we design multi-dimensional BMQ-Index to support various applications requiring multi-dimensional semantics.

This paper is organized as follows. Section 2 introduces border monitoring scenarios and discusses BMQ semantic. One-dimensional BMQ-Index is presented in Section 3, and multi-dimensional version is presented in Section 4. Section 5 presents experimental results. Section 6 discusses related work. Finally, we conclude the paper.

2. Border monitoring query

In this section, we characterize a new range query semantic, namely *border monitoring* and show its importance and usefulness in stream-based applications.

Many stream-based applications continuously monitor a large number of data streams with range queries. In this situation, it is quite important to know which data streams begin or end to satisfy range conditions of queries. It is mainly because users are usually interested in whether the continuous data streams satisfy range conditions or not. Thus, it is sufficient for people to know only the beginnings and ends of satisfying range conditions rather than all satisfying events. (see scenario 2) Also, the beginnings and ends of satisfying range conditions are useful to automatically trigger or stop necessary actions. (see scenario 1 and 2) In processing standpoint, notifying only the beginnings and ends rather than all satisfying events saves network bandwidth.

2.1. Border monitoring scenarios

Scenario 1: Financial Trading
Consider the case of NASDAQ. Thousand of companies generate the streams of updates such as stock prices every 30 seconds. In addition, millions of stock investors monitor them by registering their own queries. Assume that a stock investor wants to automatically buy IBM stock right after the price of stock falls below $40 and sell his stock when the price rises above $50. In this situation, it is very useful for the investor to be notified whenever price goes above or below a user-specified border.

Scenario 2: Location-based Advertisement
As shown in Figure 1, many restaurants, cafes, and gas stations are willing to advertise lunch menu or send a discount coupon to people within nearby rectangle regions for two hours. For this service, it is required to quickly locate the people who are coming into or going out from the specified region by monitoring the streams of people's locations. People do not like to receive the same advertisement more than once. Thus, it is not necessary to locate the people which are already in the region.

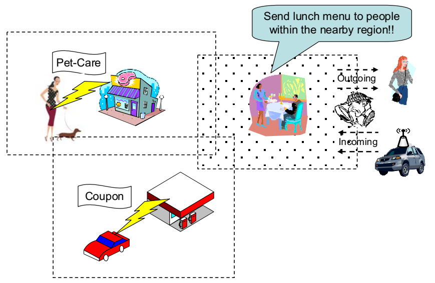


Figure 1. Location-based Advertisement

2.2. Semantic definition of BMQ

The semantic of range queries in the above border monitoring scenarios are different from well-known range queries in data stream processing. To distinguish such queries, we classify continuous range queries into two types, i.e., *Border Mointoring Queries (BMQ)* and *Region Monitoring Queries (RMQ)*. Continuous range queries in existing data stream processing fall into the category of RMQ, i.e., the query that reports all data within a query range. BMQ is a type of query which reports only data crossing the border of a query range.

The formal definition of BMQ on the set of data is as follows. Suppose that two consecutive sets of data and a BMQ are given. Let $RSet(t–1)$ represent the data contained in the query range at previous update time $t–1$ and $RSet(t)$ represent the ones at current update time t. Then, two sets of data are defined as the result of the BMQ.

***Definition 1.* Border Monitoring Query (BMQ)**

- $RSetBMQ^{+}(t) = RSet(t) - RSet(t-1)$
- $RSetBMQ^{-}(t) = RSet(t-1) - RSet(t)$

3. One-dimensional BMQ-Index

We first outline the main idea of BMQ-Index, *shared and incremental processing*.

- *Shared processing*

In border monitoring scenarios, a large number of BMQs can be issued by users. In order to achieve a high level of scalability, shared processing of BMQs is essential. For this purpose, BMQ-Index adopts a query indexing approach. Once BMQ-Index is built on registered BMQs, only relevant queries are quickly searched for without unnecessary access to irrelevant queries.

Upon an incoming data tuple[1], BMQ-Index retrieves two sets of relevant queries: (1) $QSet^{+}(t)$, the set of queries that match the current data value v_t, but do not

[1] We assume that a tuple in a data stream has three attributes: *stream_ID* (the ID of stream source), *value* (the value measured at the stream source), and *time_stamp* (the time when the value was measured)

match the previous data value v_{t-1}[2]. (2) $QSet^{-}(t)$, the set of queries that do not match the current data value, but match the previous data value. We call them *differential query sets.*

- *Incremental processing*

Evaluating BMQs over continuous data streams involves successive retrievals of differential query sets. These successive operations result in considerably high processing cost when a huge volume of data streams are rapidly incoming.

To accelerate such successive operations, BMQ-Index employs an incremental access method. First, BMQ-Index stores only delta query information. It divides a *domain space*, the range of possible data values, into region segments by the borders of queries. Then, it stores a query into only two segments where the query range starts and ends[3]. We call the query stored in each segment *delta query*. Second, BMQ-Index incrementally retrieves differential queries through linear traversals from a previous matching segment to a current matching segment. Note that differential queries are easily derived from the delta queries of the visited segments.

Based on the incremental access method, successive BMQ evaluations are greatly accelerated. Due to the locality of data streams[4], an updated data tuple probably remains in the same segment, which involves only a simple comparison operation. Even if it does not, it is highly possible that an updated data tuple falls in a nearby segment. Thus, differential queries are quickly searched for with a small number of segment visits.

3.1. Index structure

BMQ-Index consists of two data structures: a *stream table* and an *RS*(*Region Segment*) *list* (see Figure 2). The stream table maintains a node pointer to the last located RS node for each data stream. A data stream is distinguished by Stream_ID although data streams simultaneously flow into BMQ-Index from multiple sources. Such identification is quickly done in $O(1)$ because the stream table entries are hashed by Stream_ID.

RS list is defined as follows. Let $Q = \{Q_i\}$ be a set of continuous range queries where a query Q_i has the range (l_i, u_i) and let B denote the set of lower and upper bounds of the range of each Q_i in Q, i.e., $B = \{b \mid b \text{ is either } l_i \text{ or } u_i \text{ of a } Q_i \in Q\} \cup \{\infty\}$. We denote the elements of the set B with a subscript in increasing order of their values. That is, $b_0 < b_1 < \ldots < b_m < b_{m+1}$.

[2] v_{t-1} and v_t are the values of two consecutive tuples of a data stream.

[3] We do not store an entire query string in the segment, just a query ID.

[4] In our previous work [9][10], we demonstrate the existence and characteristic of the locality in real data streams.

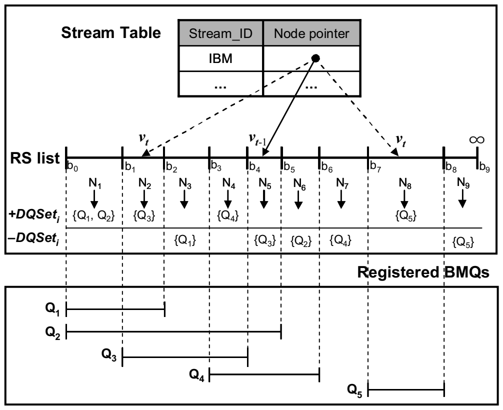


Figure 2. Structure of BMQ-Index

An RS list is a list of RS nodes, $<N_1, N_2, \ldots, N_m, N_{m+1}>$. Each RS node N_i is a tuple $(R_i, +DQSet_i, -DQSet_i)$, where

- R_i is the range of region segment (b_{i-1}, b_i), $b_i \in B$
- $+DQSet_i$ is the set of queries Q_k such that $l_k = b_{i-1}$ for the range (l_k, u_k) of Q_k
- $-DQSet_i$ is the set of queries Q_k such that $u_k = b_{i-1}$ for the range (l_k, u_k) of Q_k

An RS node holds two delta query sets, $+DQSet_i$ and $-DQSet_i$. $+DQSet_i$ is the set of queries Q_k that share the lower bound of their range with that of R_i, i.e., $Q_k \in +DQSet_i$ if $l_k = b_{i-1}$. Similarly, a query Q_k belongs to the $-DQSet_i$ of an RS node N_i if the upper bound of its range forms the lower bound of R_i, i.e., $Q_k \in -DQSet_i$ if $u_k = b_{i-1}$.

In Figure 2, an RS list is built for five BMQs. Nine RS nodes are created. Each node has a range and $\pm DQSet_i$. For instance, N_5 has a range (b_4, b_5), $\{\}$ as a $+DQSet_5$, and $\{Q_5\}$ as a $-DQSet_5$.

3.2. Query registration and deregistration.

A query can be dynamically registered and deregistered in BMQ-Index. Assume that a query Q_{in} whose range is (l_{in}, u_{in}) is registered. First, BMQ-Index locates the RS node, N_i which contains l_{in}, i.e., $b_{i-1} \le l_{in} < b_i$. If l_{in} is equal to b_{i-1}, Q_{in} is inserted into the $+DQSet_i$ of N_i. Otherwise, N_i is split into two RS nodes: the left node with the range of (b_{i-1}, l_{in}) and the right node with the range of (l_{in}, b_i). The left node has the $\pm DQSet$ of N_i, and the right node contains Q_{in} in its $+DQSet$. Second, BMQ-Index locates the RS node, N_j which contains u_{in}, i.e., $b_{j-1} \le u_{in} < b_j$. If u_{in} is the same as b_{j-1}, Q_{in} is inserted into the $-DQSet_j$ of N_j. Otherwise, N_j is also split into the two RS nodes: the left node with the range of (b_{j-1}, u_{in}) and the right node with the range of (u_{in}, b_j). The left node has the $\pm DQSet$ of N_j, and the right node keeps Q_{in} in its $-DQSet$.

When a query Q_{out} whose range is (l_{out}, u_{out}) is deregistered, BMQ-Index first locates the RS node, N_i whose lower bound is equal to l_{out}, and removes Q_{out} from the $+DQSet_i$. If both $+DQSet_i$ and $-DQSet_i$ are empty, N_i

is merged with N_{i-1}. Second, BMQ-Index locates the RS node, N_j whose lower bound is u_{out}, and removes Q_{out} from $-DQSet_j$. If both $+DQSet_j$ and $-DQSet_j$ are empty, N_j is merged with N_{j-1}.

3.3. Incremental search algorithm

In BMQ-Index, differential query sets are efficiently retrieved from delta query sets. Given two consecutive data values, v_{t-1} and v_t, let v_{t-1} fall in the range of an RS node N_j and v_t fall in that of N_h, i.e., $b_{j-1} \le v_{t-1} < b_j$ and $b_{h-1} \le v_t < b_h$. While visiting from N_j to N_h, two differential query sets, $QSet^+$ and $QSet^-$ are evaluated as follows.

$$\begin{aligned}
&\text{If } j < h,\ QSet^+ = [\textstyle\bigcup_{i=j+1}^{h} +DQSet_i] - [\bigcup_{i=j+1}^{h} -DQSet_i]\\
&\qquad\quad QSet^- = [\textstyle\bigcup_{i=j+1}^{h} -DQSet_i] - [\bigcup_{i=j+1}^{h} +DQSet_i]\\
&\text{If } j > h,\ QSet^+ = [\textstyle\bigcup_{i=j}^{h+1} -DQSet_i] - [\bigcup_{i=j}^{h+1} +DQSet_i]\\
&\qquad\quad QSet^- = [\textstyle\bigcup_{i=j}^{h+1} +DQSet_i] - [\bigcup_{i=j}^{h+1} -DQSet_i]\\
&\text{If } j = h,\ QSet^+ = QSet^- = \phi
\end{aligned}$$

The evaluation of $QSet^+$ and $QSet^-$ depends on the relative order between N_j and N_h. If $j < h$, $QSet^+$ has all queries in the union of $+DQSet_i$ excluding the queries in the union of $-DQSet_i$ where i takes the values from $j+1$ to h. Similarly, $QSet^-$ is calculated by subtracting the union of $+DQSet_i$ from the union of $-DQSet_i$. On the other hand, if $j > h$, $+DQSet_i$ and $-DQSet_i$ are switched while i takes the values from j to $h + 1$. There is no differential query if $j = h$. (See Appendix **A.2** of [9] for the proof).

Figure 2 shows the examples of our incremental search algorithm. Assume that the previous data value v_{t-1} was located in N_5. If the current data value v_t is located in N_8, $\pm DQSet$ are retrieved while visiting from N_6 to N_8. Thus, $QSet^+ = \{Q_5\}$ and $QSet^- = \{Q_2, Q_4\}$. If v_t is located N_2, $\pm DQSet$ are retrieved during node visits from N_5 to N_3. Thus, $QSet^+ = \{Q_3, Q_1\}$ and $QSet^- = \{Q_4\}$.

3.4. Analysis of search and storage cost

The search cost of BMQ-Index can be represented as the total number of retrieved delta queries. The average number of retrieved delta queries U is determined by two factors. First, *U is proportional to the average distance between two consecutive data values*. As the distance increases, more RS node visits are required to locate a new matching node, thereby increasing the number of retrieved delta queries. We define *Fluctuation Level* (*FL*) as the average distance normalized with respect to the domain size.

$$FL = \frac{\text{Average distance}}{\text{Domain size}} = \frac{\sum_{i=2}^{M} |X_i - X_{i-1}|}{M - 1} \times \frac{1}{\text{Domain size}}$$

(X_i is ith data value and M is the total number of data tuples)

Second, *U is proportional to the average density of delta queries in an RS list*. As the density increases, more delta queries are retrieved even with the same FL. The average density of delta queries in an RS-list can be approximated as ($2 \times N_q$ / Domain size), where N_q is the number of BMQs. It is because each query ID is inserted only twice into an RS list. Thus, the average search cost of BMQ-Index can be formulated as $\Theta(2 \times N_q \times FL)$.

The storage cost of BMQ-Index is decided by the sizes of an RS list and a stream table. The size of the RS list is $\Theta(2N_q)$ since each query is inserted once into $+DQSet$ and $-DQSet$, respectively. The size of the stream table is proportional to the number of input data streams, N_d. Consequently, the total storage cost of BMQ-Index is $\Theta(2N_q + N_d)$.

4. Multi-dimensional BMQ-Index

We design multi-dimensional BMQ-Index by directly extending one-dimensional BMQ-Index. N-dimensional BMQ-Index stores delta query information in N different RS lists. Each RS list contains borders and delta queries for one of N dimensions. Upon a data arrival, all RS lists are searched in order to obtain differential query sets per dimension. We develop an efficient cross-check algorithm, which validates queries in the per-dimension differential query sets to identify a final result.

Our solution approach has three advantages. First, it has significantly low storage cost. It is because a query is not repeatedly saved in an N-dimensional region but saved only a few times in N one-dimensional RS lists. Note that a query is saved only twice in an RS list. Second, it has an high search performance. As shown in Section 4.4, the search algorithm including the cross-check requires only $(N-1)\sqrt{N}$ times of search time for one-dimensional BMQ-Index. For the two-dimensional index, only $\sqrt{2}$ times as much search time as the one-dimensional index is needed. Finally, multi-dimensional BMQ-Index can be easily implemented due to the simplicity of the index structure and its access algorithms.

In the rest of this section, we present two-dimensional BMQ-Index to simplify the explanation.

4.1 Index structure

Two-dimensional BMQ-Index consists of following data structures: two RS lists (an RS-X list and RS-Y list), a stream table, and a query table. Figure 3 shows an example of the index with three registered queries. The RS-X list is a list of region segments that together comprise the range of an X-dimension, <$RS\text{-}X_1$, $RS\text{-}X_2$, …, $RS\text{-}X_n$>. Each region segment $RS\text{-}X_i$ maintains lower and upper bounds of the region and $\pm DQSet$ for the X-dimension. The RS-Y list maintains the information for a Y-dimension similar to the RS-X list.

Stream Table

StreamID	V	P_X	P_Y
s1	(v_{X1}, v_{Y1})	RS-X_2	RS-Y_2
s2	(v_{X2}, v_{Y2})	RS-X_3	RS-Y_5
s3	(v_{X3}, v_{Y3})	RS-X_5	RS-Y_4

Query Table

QueryID	Range
Q_1	$(b_{X1}, b_{X3}, b_{Y1}, b_{Y4})$
Q_2	$(b_{X2}, b_{X6}, b_{Y2}, b_{Y6})$
Q_3	$(b_{X4}, b_{X5}, b_{Y3}, b_{Y5})$

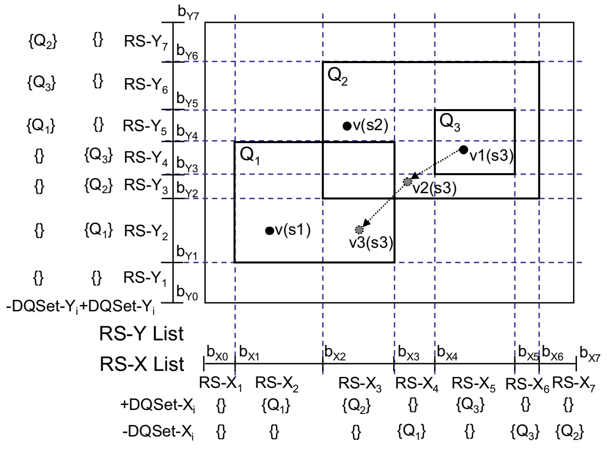


Figure 3. Two-dimension BMQ-Index

In a two-dimensional case, each entry of the stream table has two pointers, P_x and P_y, pointing *RS-X_i* which contains the current X-dimension value of the stream, and *RS-Y_i*, which contains the current Y-dimension value of the stream. Also, current data value is saved for the next search operation. The stream table entry is updated upon an arrival of a new data tuple for each data stream. The query table, which is hashed with query ID, saves borders of queries; it is required for the cross-check algorithm.

4.2 Query registration and deregistration

Two-dimensional BMQ-Index also supports dynamic query registration and deregistration. Upon a query registration and deregistration, an X-dimension predicate and Y-dimension predicate of a query are separately processed. Consider a query Q_n, whose range is (x_l, x_u, y_l, y_u). When registering Q_n to the index, an X-dimension predicate, (x_l, x_u), is registered in the RS-X list and an Y-dimension predicate, (y_l, y_u), is registered in the RS-Y list. It is done by the one-dimensional query registration method. Also, Q_n is added to the query table. Deregistration of Q_n is similarly processed using one-dimensional deregistration method.

4.3 Search algorithm

Upon an arrival of a data value, two-dimensional BMQ-Index is searched to obtain $QSet^+$ and $QSet^-$. Figure 4 shows overall flow of the search algorithm. The first step of the algorithm is to calculate differential query sets for each dimension: ±*XQSet* and ±*YQSet*. This is simply done by applying one-dimensional incremental search algorithm on the RS-X list and RS-Y list.

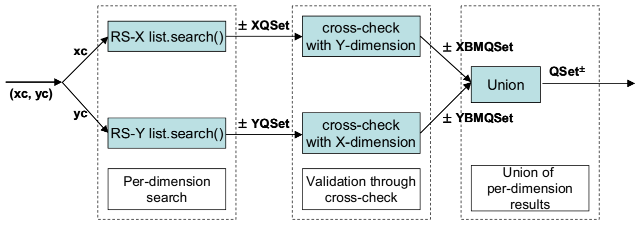

(xc, yc)
xc
yc
RS-X list.search()
RS-Y list.search()
± XQSet
± YQSet
cross-check with Y-dimension
cross-check with X-dimension
± XBMQSet
± YBMQSet
Union
QSet±
Per-dimension search
Validation through cross-check
Union of per-dimension results

Figure 4. Flow of a search algorithm

```
/* Cross-check algorithm to validate queries in ±XQSet and ±YQSet */
/* Input : stream s_i's data tuple with value of (v_Xc, v_Yc)                */

/* Initialize the result sets */
±XBMQSet= {} and ±YBMQSet = {};

/* Validate ±XQSet through cross-check with Y-dimension */
For each element Q_i of +XQSet
    Get Q_i's Y-dimension predicate, (Q_i_yl, Q_i_yh), from the query table
    If(Q_i_yl <v_Yc< Q_i_yh)   +XBMQSet ← Qi

Obtain s_i's previous data value, (v_Xp, v_Yp), using the stream table
For each element Q_i of –XQSet
    Get Q_i's Y-dimension predicate, (Q_i_yl, Q_i_yh), from the query table
    If(Q_i_yl <v_Yp< Q_i_yh)   -XBMQSet ← Qi

/* Validate ±YQSet through cross-check with X-dimension */
Cross-check queries in ±YQSet with X-dimension using above method;

/* Output : ±XBMQSet and ±YBMQSet */
```

Figure 5. A cross-check algorithm

The second step is to validate if all the borders of queries in ±*XQSet* and ±*YQSet* are indeed crossed by the data value. The validation is required because those queries may not satisfy the condition for other dimension. For the validation, we developed an efficient cross-check algorithm described in Figure 5. The cross-check algorithm examines borders of unchecked dimensions of the queries in per-dimension differential query sets. For example, if a query Q_i belongs to +*XQSet*, the cross-check algorithm checks if the data value actually crosses the Y-dimension border of Q_i. A cross-check method for +*XQSet*(+*YQSet*) is different from that for –*XQSet*(–*YQSet*). For a query in +*XQSet*, it is checked if a newly arrived data value is located between the Y-dimension borders of the query. On the other hand, for a query in –*XQSet*, it is checked if the previous value of the stream was located between the Y-dimension borders.

Through the cross-check, the verified result BMQ sets, ±*XBMQSet* and ±*YBMQSet*, are obtained. Finally, $QSet^+$ is calculated as a union of +*XBMQSet* and +*YBMQSet*. $QSet^-$ is also calculated similarly.

4.4. Analysis of search and storage cost

The search performance of multi-dimensional BMQ-Index is determined by the cost of RS node visits and the cost of the cross-check. The total number of RS node visits on multiple RS lists is decided by the sum of projections of distance vector on each dimension. Under the assumption that the average distance is same as one-dimensional case, the number of RS node visits becomes

$\sqrt{d}$ times as many as that of one-dimensional BMQ-Index in maximum, where d is the number of dimensions. In the cross-check, d–1 times of comparison are performed for queries in per-dimension differential query sets, since predicates for all other dimensions should be checked. Therefore, the required search cost is $(d-1)\sqrt{d}$ times as much as that of one-dimensional BMQ-index, thereby being $\Theta((d-1)\sqrt{d}\times 2N_q \times FL)$, where N_q is the number of BMQs.

The storage consumption is decided by the sizes of RS lists, a stream table and a query table. Since there is an RS list per dimension and the stream table has an RS node pointer per dimension, the storage size for the RS lists and the stream table is d times as large as that of one-dimensional BMQ-Index. Additionally, multi-dimensional BMQ-Index maintains a query table, thus the storage cost of multi-dimensional BMQ-Index is $\Theta(d(2N_q + N_d) + N_q)$, where N_d is the number of input data streams.

5. Experiments

In this section, we present our experimental results and discuss the performance of BMQ-Index. Due to the page limitation, we only present the result for one-dimensional BMQ-Index which is enough to show the effectiveness of BMQ-Index. Interested readers can find the preliminary result for multi-dimensional BMQ-Index in [10].

We first explain how data streams and queries are generated as a workload. Next, we compare the search performance and storage cost of BMQ-Index with a mechanism based on query indices for RMQ evaluation, namely *DiffRMQ*. DiffRMQ derives differential query sets via two steps. It first retrieves consecutive matching query sets for previous and current data values. Then, it performs a set difference operation on them to remove the queries containing both data values. The state-of-the-art query indices, i.e., CEI [17] and IS-list [4], are used for DiffRMQ. The experiments are conducted using a machine equipped with P-III 1GHz CPU, 512MB RAM, and Linux 2.4.

5.1. Stream and query generation

5.1.1. Stream generation. As an experimental scenario, we consider the financial trading which is described in section 2.1. Based on the real observation on Korean stock market (see details in [9]), we synthetically generate stock price streams. We consider that FL varies from 0.01% to 0.1%, which are derived from the real traces. We use multiple stream sources as an input; the number of stream sources is 2,000 and each stream contains 1,000 data tuples, respectively.

5.1.2. Query generation. Queries specify the ranges of stock prices. The lower bounds of query ranges are randomly chosen between 1 and D – 1, where D is fixed to 1,000,000 which is the maximum price of most Korean stocks [7]. In practice, the width of query W is usually larger than FL. Thus, we regard that W is 1 ~ 10 times larger than FL. W is also normalized with respect to the domain size. Finally, the number of queries N varies from 10,000 to 100,000.

5.2. Comparison with alternative approaches

5.2.1. Average search time. In this experiment, we compare the search performance of BMQ-Index with that of DiffRMQ. For intuitive comparison, we first measure the *Search Efficiency* (*SE*) which is defined as follows.

$$SE = \frac{\sum_{i=1}^{M} \text{size of final result}}{\sum_{i=1}^{M} \text{size of intermediate result}}$$

(M is the total number of input data tuples)

In BMQ-Index, the size of intermediate result is the total number of accessed delta queries during an evaluation. In DiffRMQ, it is the number of accessed matching queries. In both cases, the size of final result is the total number of differential queries after the evaluation. Then, we measure average search time to show practical search performance. We measure SE and search time while varying three parameters: the number of queries N, the width of queries W and fluctuation level FL.

First, we vary N from 10,000 to 100,000. W and FL are fixed to 0.1% and 0.01%, respectively. Figure 6 shows SE and average search time as a function of the number of queries. The SE of BMQ-Index is 100% regardless of the number of queries and much higher than the SE of DiffRMQ. In DiffRMQ, consecutive matching query sets are likely to much overlap due to the locality of data stream. Thus, many irrelevant queries are accessed to obtain only a few differential queries. Consequently, the search time of DiffRMQ is significantly higher than that of BMQ-Index. The slight increase in the search time of BMQ-Index mainly comes from the increase in the final result size as the number of registered queries increases.

Second, we vary W from 0.01% to 0.1%. N and FL are fixed to 100,000 and 0.01%, respectively. Figure 7 shows the results. As the width of queries increases, the SE of DiffRMQ rapidly decreases, whereas that of BMQ-Index remains almost 100%. It is because the size of intermediate result of DiffRMQ increases as the width of queries increases. However, that of BMQ-Index is not affected by the width of queries. Note that the size of final result does not change in both cases. Therefore, as the width of queries increases, the search time of DiffRMQ increases significantly, and that of BMQ-Index is kept as a constant.

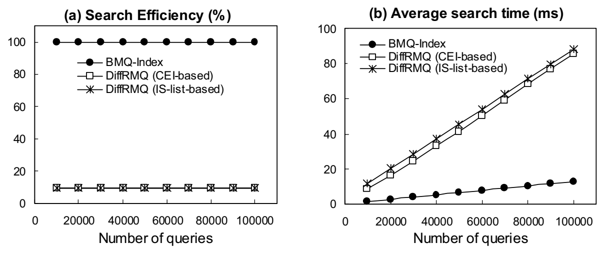


Figure 6. Effect of the number of queries (W=0.1%, FL=0.01%)

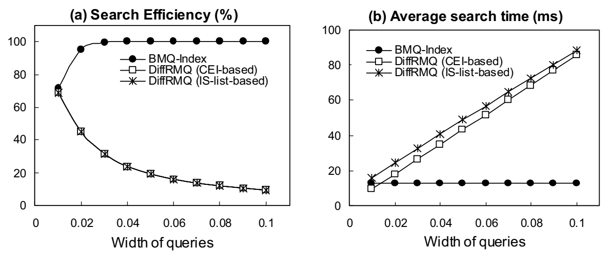


Figure 7. Effect of the widths of queries (N=100000, FL=0.01%)

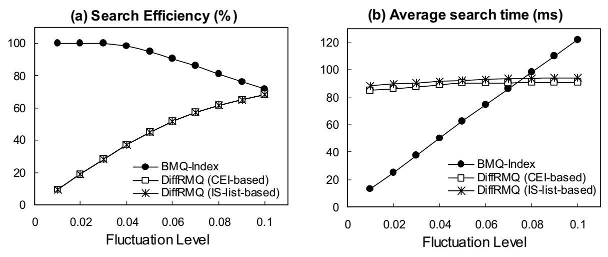
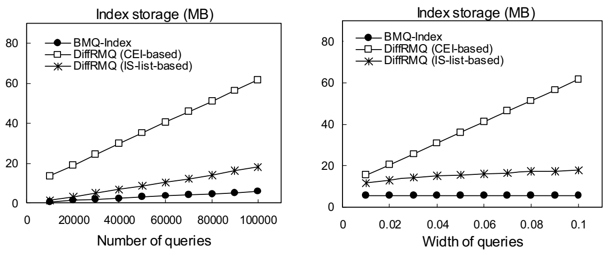


Figure 8. Effect of the Fluctuation Level (N=100000, W=0.1%)

Finally, we vary FL from 0.01% to 0.1%. N and W are fixed to 100,000 and 0.1% respectively. As FL increases, the size of intermediate result of BMQ-Index increases while that of DiffRMQ does not change. Also, the size of final result increases in both cases as FL increases. Thus, the SE of BMQ-Index decreases slowly while that of DiffRMQ increases rapidly as shown in Figure 8 (a).

Interestingly, the SE of BMQ-Index is higher than that of DiffRMQ as long as FL is smaller than W. Consequently, the search time of BMQ-Index is much smaller than that of DiffRMQ when FL is smaller than W. Note that W is relatively larger than FL in practice. If FL increases up to W, the search time of BMQ-Index is slightly larger than that of DiffRMQ due to additional cost for RS node traversals.

5.2.2. Storage cost. In this experiment, we compare the storage cost of BMQ-Index with that of DiffRMQ. We measure the memory space taken by each query index. To identify the effect of N and W on the storage cost, we run two experiments. In the first experiment, we vary N from 10,000 to 100,000 and fix W to 0.1%. In the second, we vary W from 0.01% to 0.1% and fix N to 100,000.



(a) The effect of N **(b) The effect of W**

Figure 9. Storage cost

Figure 9(a) shows the size of index storage as a function of the number of queries. BMQ-Index uses much less memory than two DiffRMQ approaches, especially than CEI-based DiffRMQ. In general, query indices for RMQ evaluation store queries redundantly. CEI stores a query into many grids covered by the range of query. Even tree-based query index, i.e., IS-list, stores a query log (*the number of registered queries*) times. In contrast, BMQ-Index stores a query only twice, thereby showing a lot better storage usage with the same number of queries.

Figure 9(b) shows the size of index storage as a function of the width of queries. The storage cost of CEI-based DiffRMQ increases rapidly as the widths of queries increase. In CEI, a query with a large width is repeatedly inserted in the grids overlapping the query range, resulting in high storage cost. The storage cost of IS-list-based DiffRMQ is less affected by the width of queries because it has tree-based structure. However, BMQ-Index shows a constant storage usage regardless of the width of queries since it only stores two delta queries for a query registration.

6. Related Work

Semantic of BMQ: In the context of query languages for data streams, a concept of I-Stream and D-Stream operators has been proposed [1]. I-Stream operator retrieves the set of data tuples which are inserted into a Relation, and D-Stream operator retrieves the set of data tuples which are deleted from a Relation. They represent a general concept to transform a Relation to a Stream. On the other hand, the BMQ is a specific class of continuous range query, and it has useful meanings in practical stream-based applications as described in section 2.1.

Shared and incremental processing: In the context of data stream processing, there have been extensive researches on evaluating a large number of continuous range queries. However, they concentrate on Region Monitoring Query (RMQ) rather than Border Monitoring Query (BMQ).

Query indexing is widely used for shared evaluation of RMQs. We call it RMQ-Index. Upon each data arrival, matching queries are quickly determined by searching the query index. Existing RMQ-Index can be classified into one-dimensional indices [2][4][5][14][17] and two-dimensional indices [6][8][13][16]. For 1-D range queries, two different approaches are proposed: tree-based query indices [2][4][5][14] and a grid-based query index [17]. The tree-based indices have $O(\log N)$ search cost and $O(N \log N)$ storage cost, where N is the number of registered queries. Compared to the tree-based indices, the grid-based query index has better search performance. Those two approaches are also used to support 2-D range queries. 2-D grid-based indices [8][16] show much better search performance than the 2-D tree-based indices [6][13]. However, the grid-based indices require much more index storage since queries are redundantly inserted into many grids covered by query ranges.

Due to the semantic difference between RMQ and BMQ, the existing RMQ-Indexs are generally not suitable for BMQs. Even though the RMQ-Index can be used for BMQ evaluation, it requires costly post-processing to identify only the data streams crossing the borders of the queries. Thus, the performance is considerably low compared to that of BMQ-Index which is designed primarily for efficient BMQ evaluation.

In the context of spatio-temporal database, SINA [11] and GPAC [12] have been proposed for the incremental evaluation of RMQs. Similar to BMQ-Index, they compute only updates from the previously reported answer (positive and negative updates), thereby reducing the processing overhead of continuous reevaluation. However, GPAC is designed for an evaluation of one outstanding continuous query, not for shared processing of multiple queries. To achieve shared processing, SINA performs a spatial join between a set of objects and a set of queries. However, SINA adopts a disk-based algorithm so that SINA is difficult to be applied in data stream processing in which in-memory processing is essential. Furthermore, SINA is not a purely incremental access method. SINA stores the query information redundantly based on a grid index, rather than storing delta query information. Thus, complex invalidation as well as spatial join is required to retrieve the positive and negative updates. In contrast, BMQ-Index retrieves them with an incremental search operation.

7. Conclusion

In this paper, we first emphasize the importance and usefulness of BMQ. Then, we propose BMQ-Index, which evaluates a large number of BMQs efficiently in a shared and incremental manner. We demonstrate the excellent search performance and low storage cost of BMQ-Index through analysis and experiments. Currently, we are extending BMQ-Index to support aggregation and join operation and developing a specialized system for border monitoring services.

8. References

[1] A. Arasu, S. Babu and J. Widom, "The CQL Continuous Query Language: Semantic Foundations and Query Execution" *Technical Report* Oct. 2003

[2] S. Chandrasekaran and M. J. Franklin, "Streaming Queries over Streaming Data", *VLDB* 2002

[3] L. Golab and M. Tamer Ozsu, "Data Stream Management Issues – A Survey", *SIGMOD Record* 2003

[4] E. Hanson and T. Johnson, "Selection Predicate Indexing for Active Databases using Interval Skip Lists", *Information Systems*, 21(3):269–298, 1996

[5] E. Hanson, M. Chaabouni, C. Kim, and Y. Wang, "A predicate matching algorithm for database rule systems", *SIGMOD* 1990

[6] H. Hu, J. Xu and D. Lee, "A Generic Framework for Monitoring Continuous Spatial Queries over Moving Objects", *SIGMOD* 2005

[7] Korea stock exchange. http://www.kse.or.kr.

[8] D. V. Kalashnikov, S. Prabhakar, W. G. Aref, and S. E. Hambrusch, "Efficient evaluation of continuous range queries on moving objects", DEXA 2002

[9] J. Lee, S.Kang, S. Choi, H. Jin, S. Choe, and J. Song, " LARI: Locality-Aware Range query Index for High Performance Data Stream Processing", *Technical Report* August 2004 available in http://nclab.kaist.ac.kr/~jcircle/publication.html

[10] S. Lee, S. Kang, J. Lee, Y. Lee, B. Kim, H. Jin, and J. Song, "M-LARI: Locality-Aware Multidimensional Range query Index for Border Monitoring Queries over Data Streams", *Technical Report,* January 2005 available in http://nclab.kaist.ac.kr/~jcircle/publication.html

[11] M. F.Mokbel, X. Xiong and W. G. Aref, "SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Database", *SIGMOD* 2004

[12] M. F.Mokbel and W.G.Aref, "Generic and Progressive Processing of Mobile Queries over Mobile Data", *MDM* 2005

[13] S. Prahakar, Y. Xia, D. V.Kalashnikov, W. G.Aref and S. E. Hambrusche. "Query In-dexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects", *IEEE Transaction on Computers* 51(10):1124-1140, 2002

[14] S. R. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman, "Continuously Adaptive Continuous Queries over Streams", *SIGMOD* 2002

[15] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "The Design of an Acquisitional Query Processor for Sensor Networks", *SIGMOD* 2003

[16] K.L. Wu, S. Chen and P. S. Yu, "Indexing Continual Range Queries for Location-Aware Mobile Services", *EEE* 2004

[17] K. L. Wu and P. S. Yu, "Interval Query Indexing for Efficient Stream Processing", *CIKM*, 2004