BMQ-Processor: A High-Performance Border-Crossing Event Detection Framework for Large-scale Monitoring Applications

Jinwon Lee, Seungwoo Kang, Youngki Lee, SangJeong Lee, and Junehwa Song

Abstract—In this paper, we present *BMQ-Processor*, a high performance border-crossing event detection framework for large-scale monitoring applications. A border monitoring query (BMQ) is useful for border-crossing event detection in many monitoring applications. It monitors the values of data streams and reports them only when data streams cross the borders of its range. BMQ-Processor efficiently handles a large number of border crossing events over a high volume of data streams. It develops and operates over a stateful query index, achieving a high level of scalability over continuous data updates. Also, it utilizes the locality embedded in data streams and greatly accelerates successive BMQ evaluations. We present data structures and algorithms to support one-dimensional as well as multi-dimensional BMQs. We show that the semantics of border monitoring can be extended toward more advanced ones and build region transition monitoring as a sample case. Lastly, we demonstrate excellent processing performance and low storage cost of BMQ-Processor through extensive analysis and experiments.

Index Terms—Database semantics, Indexing methods, Mobile environment, Query processing, Sensor network.

1 Introduction

Recent advances in mobile computing and embedded device technologies open up new opportunities for various types of advanced monitoring applications, e.g., location-aware [36][46], context-aware [6][18], environmental [45] and financial [2][15] monitoring applications. An important feature of such applications lies in situation-awareness; the applications continuously monitor and identify if situations or conditions of interest occur. Services are automatically triggered upon the detection of the registered conditions, e.g., an air conditioner is automatically turned on if the temperature of an office is higher than 28 °C. Much research on active databases and event-based systems has been performed to support such event detection and task automation [11][20][21][26][32][50][56][58]. The event detection is done through continuous monitoring of numerous data streams generated from various sensors, GPSs, or agents that are widely deployed throughout physical or virtual (computing) environments. Often, such monitoring applications are large-scale, spanning a number of people and devices over a wide geographic area. An efficient event detection framework is necessary to effectively support large-scale monitoring applications.

An important class of events in large-scale monitoring applications is the border-crossing event (BCE). A BCE is intuitively represented as a data stream crossing the borders of a user-specified interest range. Note that this semantics is different from that of the commonly used range filtering which reports all data matching to a specified range condition [2][46][32]. BCE processing is motivated by two observations. First, for many monitoring applications, it is sufficient to report to users only the triggering and stopping events of a user-specified range condition, rather than to report all matching data. Second, the events are frequently accompanied by actions beyond monitoring itself. They are compelling to users who want the appropriate actions to be automatically triggered or stopped. (See example scenarios in Section 3) Recently, the border-crossing concept has been also recognized as an important way to detect events in the field of sensor networks [1][31][56][65], and will be essential to enable emerging action-oriented actuator networks [42][55] coupled with sensor networks to operate automatically.

In this paper, we present BMQ-Processor, a high performance border-crossing event (BCE) detection framework for large-scale monitoring applications. A high performance framework is important for large-scale monitoring applications, especially in the emerging sensor-rich mobile and pervasive environments; it should handle a high volume of data streams continuously arriving from a number of sources. In addition, individual users issue different requests, personalized to their own needs, resulting in numerous user requests. Furthermore, they expect real-time responses and are not tolerant of stale events and delayed responses. To develop the framework, we take note of the practical importance of BCEs, especially the massive processing of BCEs, and characterize query semantics, namely Border Monitoring Query

- *The authors are with the Department of Computer Science, KAIST, 373-1 Gusong-dong Yusong-gu Daejon 305-701 Republic of Korea. (telephone: +82-42-869-3586, e-mail: {jcircle, swkang, youngki, peterlee, junesong}@nclab.kaist.ac.kr).*
- *The preliminary version* [40] *was presented in 7th International Conference on Mobile Data Management (MDM'06), Nara Japan, May 2006, and received the* ***Best Paper Award.***

Manuscript received June 16, 2007.

(BMQ). A BMQ specifies a set of BCEs in relation to an interest range in a data-centric manner.

Our study is the first attempt to develop a high-performance BMQ-Processor which can handle a large number of BCEs over numerous high-rate data streams. DSMSs (Data Stream Management Systems) [2][14][15][44][48] have recently been receiving a lot of attention, focusing on generic system-level abstractions and performance optimizations for stream-based monitoring applications. However, event detection was not the main concern in this context and has not been studied extensively. The focus of DSMSs was on supporting continuous queries, i.e., the extension of relational query language for continuous execution. The semantics of BMQ is different from the existing continuous range query, namely the Region Monitoring Query (RMQ) [2][4][15][25][44][46], which reports all data matching a specified query range.

Event processing has been an important issue for a long time in many areas, including active databases, event-based systems, sensor networks, etc. Most notably, active databases [10][11][12][21][22][23][24][30][50][59] and event-based systems [7][16][26][32][33][58][64] have evolved for diverse application domains, e.g., logistics, surveillance and facility management, business to business integration, healthcare. In the course of these efforts, the ECA model has been established. The ECA-based systems are powerful in expressing and processing composite events, thereby enabling applications to be equipped with active capabilities [9]. Reflecting these efforts, the concept of the trigger was implemented in most of the commercial DBMSs [29].

Event-based systems are expressive enough to specify BCEs. However, they are still premature to support large-scale system environments. As mentioned, such systems should be able to effectively handle a massive amount of events, requiring a highly scalable processing framework. The trigger mechanisms in DBMSs are limited in scale with regards to supporting large-scale applications; only a few triggers per table are allowed [2][29]. To improve the performance of event processing, there have been various efforts, e.g., predicate indexing [27][28][32], Rete algorithm [19], lazy evaluation of composite events[33], sub-graph merging [17], selection mode support [12]. The techniques worked good enough to efficiently detect events and evaluate conditions in their respective target environments. However, in developing the techniques, the number of registered events as well as the input rates have not been assumed to be very high, e.g., compared to those for data stream processing [20][35]. The input rates would increase significantly in the upcoming mobile and pervasive environments [6][36][45][46]. Numerous high-rate sensors, GPSs, or agents will be increasingly deployed in surrounding spaces or even on the Internet. BMQ-Processor targets such an emerging environment with numerous data streams of potentially high data rates and a large number of queries specified on them.

To address these challenges, BMQ-Processor develops a shared and incremental processing mechanism. For shared processing, BMQ-Processor adopts a query indexing approach, thereby achieving a high level of scalability. Once BMQ-Processor is built on registered queries, only relevant queries are quickly searched for upon an incoming data. The main innovation of BMQ-Processor compared to previous approaches is that BMQ-Processor develops and operates over a *stateful index*. Existing query indices are stateless and optimized only for one-time searching. However, for data stream processing, it is extremely important to optimize the index for consecutive searching since the query index is repeatedly searched as data continuously arrives. The proposed BMQ-Processor holds the state of the last evaluation. It is structured so that, upon a new data input, the evaluation is efficiently done by starting the operation from the last state.

For incremental processing, BMQ-Processor utilizes the locality of data streams. Data updates usually exhibit gradual changes more often than abrupt ones. (See Appendix A for the study of locality in data streams.) Thus, in many cases, the matching query set for a data update will be equal to or overlap much with that for the previous update. To fully utilize this fact, BMQ-Processor calculates the difference of matching queries in advance and accordingly partitions a domain space. Upon data arrival, evaluation can be quickly done by simply traversing a small number of the partitioned segments without any complicated computation.

BMQ-Processor has two important features: excellent processing performance and low storage cost. As mentioned before, the shared and incremental processing enables BMQ-Processor to achieve remarkable processing performance. It is also superior in storage cost, storing only the difference of matching queries which consumes a small size of memory space. Note that such low storage cost is essential in large-scale stream processing where only in-memory algorithms are practical. Compared to the straightforward approach based on state-of-the-art RMQ evaluation mechanisms, BMQ-Processor achieves much better processing performance and storage cost.

Our research can be understood as a step to bridge and combine two independently evolved research efforts, i.e., data stream processing and event processing. We believe that they well complement each other to meet semantic and processing requirements of emerging monitoring applications. An interesting research can be found on this line in EStream [20][35], which envisioned the necessity of combining the two domains ahead. EStream is designed to detect composite events from data streams by sequentially connecting a data stream processing engine with an event-based system. Thereby, it combines the complex query processing capability of stream processing engines with the composite event expression and detection capability of event-based systems. Similarly, BMQ-processor extracts a meaningful pattern of data, i.e., border crossing, as important events from high-rate data streams, and further elaborates on performance issues. Using these works as a basis, we believe research on stream processing can be extended to defining and processing various events. On the other hand, research on event processing can be enriched by defining new event semantics on data streams and designing high performance processing techniques.

The contribution of this paper is summarized as follows. We develop BMQ-Processor which evaluates a large number of BMQs in a shared and incremental manner, thereby achieving excellent processing performance and low storage cost. Based on the main idea, we design a one-dimensional as well as a multi-dimensional BMQ-Processor to support various monitoring applications. In addition, we show that the semantics of border monitoring can be further extended toward more advanced ones and investigate region transition monitoring as a sample case.

This paper is organized as follows. Section 2 discusses related work. Section 3 introduces border monitoring scenarios and discusses the BCE and BMQ semantics. The one-dimensional BMQ processing is presented in Section 4, and the multi-dimensional version is presented in Section 5. Section 6 introduces the region transition monitoring and discusses its semantics and processing mechanism. Section 7 presents experimental results. Finally, we conclude the paper in Section 8.

2 Related Work

Our high-performance BMQ-Processor for large-scale monitoring applications is related to diverse research domains, i.e., active databases, event-based systems, publish/subscribe systems, data stream processing, and sensor networks. We review the many important contributions in these areas and compare them to our work. We discuss the differences between related works and ours in terms of system inputs, event/query semantics, and processing performance.

2.1 Active Databases

Throughout the 80's and 90's, extensive researches were performed to enable active capability in object-oriented, object-relational, and relational DBMSs [9]. The active features allow users to automate tasks and reduce or eliminate their interventions, e.g., alerts, integrity constraint checking, view maintenance, access control. As an underlying model for many active databases (e.g., HiPAC [10], Snoop [11][12], SAMOS [21][22], COMPOSE [23], Ode [24], Ariel [30], Starburst [59]), the ECA (Event-Condition-Action) model was conceptualized, elaborated, and widely used [50]. Based on the ECA paradigm, a large amount of academic research activities were performed, and many prototype systems were developed. A number of event specification languages were proposed along with their semantics and detection algorithms [50]. They well categorize events into primitive and composite events, and then define powerful sets of composition operators such as AND, OR, SEQ, NOT, Aperiodic, and Periodic. As a result of these efforts, trigger mechanisms were implemented in most commercial DBMSs [9][29].

Active database systems target a different system environment from the proposed BMQ-Processor. First, as system inputs, active databases mainly consider database or transaction events such as insert, delete, and update operations. Also, the trigger mechanisms in commercial DBMSs will hardly be used in the emerging large-scale monitoring applications due to their lack of scalability [2][9][29]. On the contrary, the proposed BMQ-Processor focuses on the detection of a number of BCEs over numerous data streams, e.g., location data, sensor readings. These data streams are often continuous and voluminous, requiring a high-performance processing framework.

To improve the performance of processing multiple ECA rules, several mechanisms have been proposed [19][27][28][50]. Most representatively, the Interval Skip List [27] and Interval Tree [28] develop a predicate indexing approach, i.e., building indices on multiple range conditions, which is somewhat similar to our approach. The condition predicate considered in these works retrieves all events whose attribute value falls in a given range, i.e., the RMQ concept described in Section 1. Due to the semantic difference, these indices are generally not suitable for BMQ processing. The performance benefit of BMQ-Processor is demonstrated in the experiment section.

2.2 Event-based Systems

Starting from active databases, event-based systems (e.g., IRules [7][58], CompAS [32][33], Ready [26], Cayuga [16], SASE [64]) have evolved and been expanded for diverse application domains, e.g., logistics, surveillance and facility management, enterprise applications, and healthcare. An excellent overview on the evolution of active capability toward various event-driven applications can be found in [9]. The event-based systems are powerful in expressing and detecting diverse user-defined composite events. As primitive events, they usually consider events generated from a specific application domain, e.g., online transaction logs [7][58], built-in-sensor reporting in a building [32][33] and RFID readings in a market [64]. However, the result of a complex computation on data streams is not considered as a primitive event, which is different from our viewpoint. In addition, they need to be further matured for performance to effectively handle emerging large-scale environments.

Based on the ECA paradigm, several complex event specification languages have been proposed [7][33][64], providing a rich set of operators to specify the semantics of various composite events. They extend the languages developed for active databases in terms of filtering capability [7] and window semantics [64]. The concept of a BCE can be specified using these languages. Using CEDL (Composite Event Definition Language) by Urban et al. [7][58], a BCE can be specified with a SEQ operator and parameter filters. The construct *E used in SAMOS [21][22] refers to the first occurrence of event E, which resembles the basic concept of a BCE, i.e., a single report at a crossing time. Also, the *duplicate* parameter for composite event specification in CompAS [33] can be used to specify the concept of a BCE. It is used to recognize the first and last duplicates of events as meaningful ones in their example scenarios.

Several mechanisms for composite event detection, e.g., Petri-nets, automata, and event trees or graphs, were originally proposed in active databases [12][22][24] and further enhanced in event-based systems [7][26][33][58][64]. These works provide performance improvement for the rather general case of composite events, whereas BMQ-Processor focuses on the massive processing of BCEs, i.e., a special class of events, which are practically important and useful.

Event-based systems incorporate several techniques to reduce processing and storage overhead for composite event detection and condition evaluations [7][12][17][32][33][58]. For efficient composite event detection, the concept of parameter context was proposed in [12], which restricts the detection of unnecessary instances of composite events by capturing application semantics. The idea of merging common sub-graphs among composite events was briefly discussed in [17] For efficient condition evaluation, Urban et al. proposed filtering features to reduce the rule-processing load by checking conditions on event parameters before rules are triggered [7][58]. Also, Hinze et al. improved the efficiency of primitive event filtering based on distribution-based predicate index [32]. They also proposed the efficient filtering of composite events through lazy evaluation [33], which avoids unnecessary primitive event detections.

Most of the previous works usually separately treat multiple composite events [7][12][22][24][26][50][58]. Such a separate processing of composite events may potentially limit the scalability required for massive processing. Processing time would significantly increase proportional to the number of registered events and input rates. Moreover, it easily requires considerable storage space to hold intermediary states of computation for each registered event, which makes in-memory processing difficult. Several researches on shared condition evaluation, e.g., predicate indexing [27][28][32], and shared composite event detection, e.g., sub-graph merging [17] can be considered as efforts to address the problem. The techniques are not apposite for shared BMQ processing, involving the evaluation of range conditions as well as state transitions (or compositions) at the same time. While the existing shared processing mechanisms handle the condition and transition evaluations separately, BMQ-Processor develops and incorporates the data structures that inherently enable the simultaneous evaluation of both range conditions and state transitions in a shared manner.

Also, BMQ-Processor significantly improves performance by developing an incremental processing technique. The incremental processing method is crucial in handling highly frequent data updates from a large number of stream sources. By considering consecutive pairs of data inputs and the locality embedded in the pairs, BMQ-Processor avoids the examination of any unnecessary range conditions as well as state transitions when state transitions are not expected to occur. In addition, a small amount of storage consumption in BMQ-Processor facilitates in-memory processing.

Research activities were also extended toward distributed event specification, semantics, and detection [9][41]. Sentinel [13] developed a well-designed global event detector, and various tools for the ease of specification of events and rules. Other works include a distributed event composition and detection framework [52] and a CORBA-based event architecture [43].

2.3 Data Stream Management Systems

For the last several years, a significant amount of progress has been made in the field of data stream processing [25]. Several data stream management systems (DSMS) such as NiagaraCQ [15], Aurora [2], TelegraphCQ [14][44], and STREAM [4][48] have been developed to enable a number of new monitoring applications. They deal with various issues such as relational continuous query languages, operator scheduling, load shedding, and fault tolerance.

There have been extensive researches on evaluating a large number of continuous range queries. However, they concentrate on processing RMQs rather than BMQs. Widely adopted mechanisms for shared evaluation of RMQs are query indices, namely RMQ-Index. RMQ-Indices have been studied for one-dimensional [14][44][63] and two-dimensional range queries [34][36][53][61][62]. The indices can again be classified into a tree-based query index ([14][44] for 1-D and [34][53] for 2-D) and a grid-based query index ([63] for 1-D and [36][61][62] for 2-D). 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 for both 1-D and 2-D. However, the grid-based indices require much larger storage space since queries are redundantly inserted into multiple grids depending on query ranges. Generally, the grid-based indices for 2-D consume larger storage space than those for 1-D due to the increase in the number of grids. The existing RMQ-Indices are limited for BMQ processing. If a RMQ-Index is used for BMQ evaluation, costly post-processing is required to sort out only the border-crossing data streams. Thus, the performance becomes considerably low compared to that of BMQ-Processor, which is specifically designed for efficient BMQ evaluation.

GPAC [47]and SINA [46] have been proposed for the efficient evaluation of RMQs over location data streams. Similar to BMQ-Processor, they compute updates from previously reported answers (positive and negative updates). However, GPAC is designed for an evaluation of a single 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 grid-based RMQ-index. Thus, as mentioned above, it involves costly post-processing to select out only the positive and negative updates from the consecutive matching sets, and incurs high storage costs. Also, SINA is a disk-based algorithm.

2.4 Sensor Networks

There have been extensive studies on sensor data monitoring and event detection in the field of sensor networks [1][3][31][56][60][65]. A typical event detection mechanism in sensor networks is to set some thresholds for sensor readings within a query [1][31][65]. It is an intuitive approach because an event in many sensor networks highly likely entails salient changes in sensor readings. In contrasting to BMQ, their monitoring queries apply typical selection semantics, i.e., RMQs, in SQL, thereby reporting all matching sensor data. Also, they mainly studied technical issues to enable efficient in-network and distributed processing of declarative queries in resource-limited sensor network environments, whereas BMQ-Processor is specialized for detecting numerous BCEs over a large number of data streams on the server side. A rule-based sensor middleware, FACTS, is proposed to enable effective high-level sensor software development [56][60]. Using the FACTS programming model,

various events on sensor data can be specified as rules. In general, we believe that BCEs can also be expressed using the FACTS rules. FACTS does not address efficient processing of a large number of registered rules.

2.5 Publish/Subscribe Systems

Publish/subscribe systems support a number of subscribers to continuously retrieve the information or events of their interest [5][8][51]. Also, information or events, e.g., RSS/news feeds, and B2B/B2C XML messages, are generated over the Internet by a number of publishers. The systems develop efficient multicast-based routing and in-network event filtering techniques considering distributed network performance.

Most systems provide filtering (e.g., point, range or keyword filtering) on primitive events and leave the task of event composition and detection to applications. Recently, PADRES [41] was proposed to enable composite event subscriptions over content-based publish/subscribe systems for application to business process and workflow integration. We think that the BCE is an important class of events also in the context of publish/subscribe systems. It could be potentially adopted to enrich the semantics of the composite events for systems like PADRES. BMQ-Processor can well be incorporated into those systems and help improve their performance.

3 Border Monitoring Query

Many monitoring applications frequently monitor a large number of data streams by specifying the ranges of interests. As discussed in Section 1, users are frequently interested in the changes in the situations rather than the details of the situations. The situation change events are also useful to automatically trigger or stop necessary actions. Moreover, notifying only on the events rather than all matching data saves computation as well as network bandwidth.

In this section, we define border-crossing events (BCEs) and border monitoring queries (BMQs) to specify the events. Also, we show its importance and usefulness in large-scale monitoring application scenarios.

3.1 Border Monitoring Scenarios

Scenario 1: Financial Trading

Consider NASDAQ. Every second, thousands of companies generate streams of updates such as stock prices, volumes, value indices, e.g., Price Earning Ratio (PER), and Price Book-value Ratio (PBR). Also, millions of stock investors monitor them by registering their own queries. Assume that a stock investor wants to do value investing. For this, he needs to continuously monitor all undervalued stocks whose prices or value indices fall below his own threshold value, e.g., PBR < 1. In this situation, it is very helpful to the investor if he is notified as soon as the data values go above or below a specified border. Based on the notifications, he can arrange his system to automatically buy or sell the stocks.

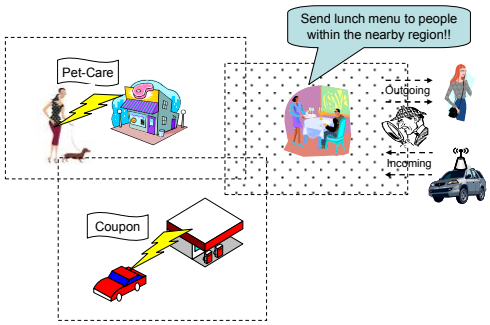


Fig. 1. Location-based Advertisement

Scenario 2: Location-based Advertisement

See Fig. 1. Many stores, like restaurants, cafes, and gas stations are willing to advertise lunch menus or send a discount coupon to people within nearby rectangle regions for about two hours. Meanwhile, the locations of people are updated every 30 seconds. People do not like to receive the same advertisement more than once. Thus, it is not necessary to locate the people who are already in the region. Instead, it is sufficient to quickly identify those who are coming into or going out of the specified region. Note that there are tens of thousands of stores in a city. Many of them would show interest in location-based advertisement to increase their profits.

3.2 Border-Crossing Event (BCE)

To clarify the implication of the BCE and help understand our insight, we first discuss the data and events in various viewpoints.

Previous researches on event-based systems and data stream management systems (DSMS) have different understandings of continuously incoming inputs, i.e., as data and as events. The subtle difference between them comes from that of the two independently developed threads of research efforts. DSMSs consider the inputs as data tuples [2][15][44][48], whereas event-based systems regard them as primitive events [16][26][32][58][64]. For example, stock feeds or sensor readings are regarded as streams of data tuples in [2][15][44] and as streams of primitive events in [16][32]. We suspect the reason is that they have targeted different types of operations. DSMSs mainly deal with relational operators such as join and aggregation, where a basic processing unit is a data tuple. On the other hand, event-based systems concentrate on various logical and temporal compositions of inputs, where a unit of composition is a primitive event.

As a bridge between the two viewpoints, we regard incoming inputs as data and meaningful patterns of data, e.g., border crossings in this paper, as primitive events. As discussed in Section 1, an interesting research is found in EStream [9][20][35], taking similar thoughts to complement the two different approaches each other. Our insight is that people are mostly interested not in raw data (e.g., each sensor reading itself), but in meaningful patterns derived from the raw data (e.g., border crossing or point of inflection on sensor readings). Previous scenarios show that border crossing is an important and frequently used pattern; whether or not the events crossing the borders of a user-specified region have occurred.

A border-crossing event (BCE) is defined formally as follows. We first define data tuples and data streams.

Definition 1. *data tuple* and *data stream*
- A data tuple is an instance of d(stream_id, value, timestamp).
- A data stream[1] is an infinite series of data tuples <d_1, d_2, d_3, … > where (d_i.stream_id = d_j.stream_id for all i, j ≥ 1) AND (d_{t-1}.timestamp < d_t.timestamp, where for all t > 1)

A BCE is defined on a data stream. It can be classified into two different types; (1) an I event representing a data stream coming into an interest range and (2) an O event representing a data stream going out from the interest range. A BCE can be described in the form of (event type, stream_id, timestamp). It can contain additional attributes, e.g., data values.

Definition 2. A *BCE* with respect to an interest range *r*
Let *s* = <d_1, d_2, d_3,…> be a data stream and *r*, an interest range. Then,
- tuple i = (I, stream_id, d_k.timestamp) is an I event for stream *s* w.r.t. *r* if there exists k, k > 1, such that (d_{k-1}.value ∉ r) AND (d_k.value ∈ r)
- tuple o = (O, stream_id, d_k.timestamp) is an O event for stream *s* w.r.t. *r* if there exists k, k> 1, such that (d_{k-1}.value ∈ *r*) AND (d_k.value ∉ *r*)

As discussed in Section 2.2, a BCE on a data stream can be expressed with several composite event specification languages, e.g., IRules [7][58], CompAS [32][33].

3.3 Definition of BMQ

A Border Monitoring Query (BMQ) is a monitoring query which collectively detects BCEs over all input data streams from a large number of sources. They handle a number of sources at the same time in a uniform way. In many sensor network and location-based systems, such an approach is importantly recognized as a data-centric paradigm [3]. Data-centric specification will proliferate as large-scale applications tend to be interested in identifying BCEs collectively regarding all data stream sources rather than the events for a specific source.

Given an interest range parameter, two sets of BMQ results are defined. $RSetBMQ^+(t)$ is the set of I events on data streams, and $RSetBMQ^-(t)$ is that of O events. They are defined through two sets of data tuples as follows. Let $q(r)$ be a query with range *r* and $RSet(t)$ be the set of data tuples which are in the range *r* at an update time *t*. Similarly, $RSet(t–1)$ represents those contained in the range at the update time *t*–1.

Definition 3. *Border Monitoring Query*
- $RSetBMQ^+(t) = RSet(t) - RSet(t–1)$
- $RSetBMQ^-(t) = RSet(t–1) - RSet(t)$

[1] A data stream corresponds to *a data source*.

Note that the definition above interchangeably used two different types of sets, i.e., the two result sets, $RSetBMQ^+(t)$ and $RSetBMQ^-(t)$, are the sets of events, whereas the other two sets, $RSet(t)$ and $RSet(t–1)$ are the those of data tuples from data streams. A more precise definition should include the type conversion of each data tuple in the result sets to an event.

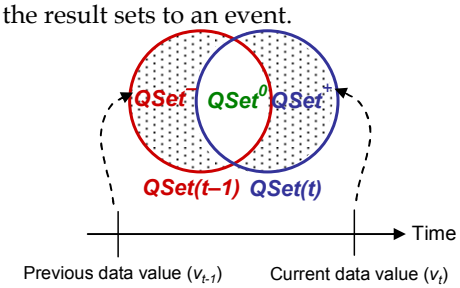


Fig. 2. Matching query sets vs. Differential query sets

4 One-dimensional BMQ-Processor

While a BMQ is effective in specifying BCEs, developing an efficient processing method is a big challenge. BMQ-Processor should handle a large number of such queries, monitoring a huge amount of data updates continuously arriving from numerous data sources. BMQ-Processor employs a shared and incremental processing mechanism to effectively deal with such query and data workloads.

In this section, we explain the key concept of the proposed method. We then present the details of one-dimensional BMQ processing, followed by an in-depth analysis on their processing and storage costs.

- *Shared processing*

To efficiently process a large number of BMQs and achieve a high level of scalability, shared processing of BMQs is essential. For this purpose, BMQ-Processor builds a query index. Once an index is built on registered BMQs, only relevant queries are quickly searched for without unnecessary access to irrelevant queries.

Upon a data tuple's arrival, BMQ-Processor retrieves two sets of relevant queries: (1) $QSet^+(t)$, the set of queries that match the current data value v_t, but do not match the previous data value v_{t-1}. (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* (see Fig. 2.). The differential query sets are defined by two matching query sets, $QSet(t–1)$ and $QSet(t)$, i.e., the matching query set of the previous data value v_{t-1} and that of the current value v_t, respectively.

Definition 4. *Differential query sets*
- $QSet^+(t) = QSet(t) - QSet(t–1)$
- $QSet^-(t) = QSet(t–1) - QSet(t)$

- *Incremental processing*

Evaluating BMQs as well as continuous range queries in general is expensive. This is especially so when the evaluation should be performed over data streams where a huge volume of data are continuously updated. To tackle the challenge, BMQ-Processor develops an incremental

processing method and significantly accelerates repeated BMQ processing. The key idea is to utilize the locality of data streams and develop a stateful query index for incremental evaluation.

Consecutive updates from a data stream usually show gradual changes. (Data may show sudden changes from time to time. However, we believe that changes are more often gradual especially in the streams of physical data. See Appendix **A** for examples. Locality in sensor data streams was also reported in the context of temporal correlation such as stair-wise and linear patterns [54][66].) Thus, in many cases, consecutive updates from each stream source fall into the regions of many of the same queries. For example, the stream [$72, $71, $73, $74] of IBM stock price falls into the region of [Q1: $70 < price < $75] at every update. We exploit such locality and the resulting overlap between matching query sets to facilitate successive BMQ evaluations. BMQ-Processor partitions a domain space into consecutive region segments, and pre-computes the differences of the sets of matching queries for consecutive segments. It then remembers the state of the last evaluation, i.e., the segment where each data stream was located at the last evaluation.

Due to the locality, an incoming data update often falls into the same segment as in the last evaluation, requiring no further evaluation. Even if it does not, it is most likely that the update falls into a nearby segment. In this case, a new evaluation is instantly done by simply taking the union of pre-computed differences. No intricate computations are involved in the process other than the union of differences. The union is taken over just a small number of consecutive segments starting from the last segment. This method is also very effective in storage cost, since it stores only the differences of queries over successive regions without replication.

4.1 Data Structure

BMQ-Processor consists of two data structures: a *stream table* and an *RS* (*Region Segment*) *list* (see Fig. 3). 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-Processor from multiple sources. The identification is quickly done in $O(1)$ because the stream table entries are hashed by Stream_ID. RS list divides a *domain space*, the range of possible data values into region segments. Each region segment of RS list holds two *delta query sets*. Given two consecutive region segments, the delta query set is the difference of matching queries for each segment.

RS list is defined as follows. Let $Q = \{Q_k\}$ be a set of continuous range queries where a query Q_k has the range (l_k, u_k) and let B denote the set of lower and upper bounds of the range of each Q_k in Q, i.e., $B = \{b \mid b$ is either l_k or u_k of a $Q_k \in Q\} \cup \{$minimum and maximum values of domain space$\}$. We denote the elements of the set B with a subscript in the increasing order of their values. That is, $b_0 < b_1 < \ldots < b_m$. An RS list is a list of RS nodes, $<N_1, N_2, \ldots, N_m>$. Each RS node N_i is a tuple $(R_i, +DQSet_i, -DQSet_i)$. R_i is the range of region segment (b_{i-1}, b_i), $b_i \in B$.

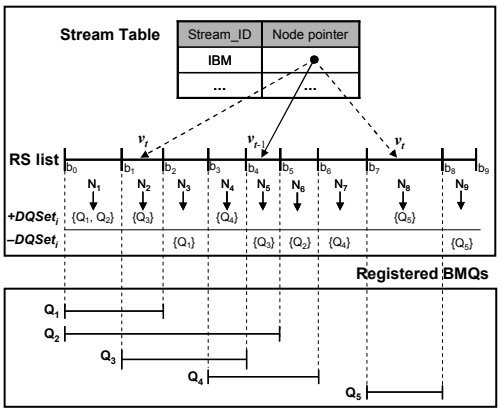


Fig. 3. Structure of BMQ-Processor

Delta query sets, $+DQSet$ and $-DQSet$, are defined as follows. Let $QSet_i$ be the set of queries matching a region segment (b_{i-1}, b_i), *i.e.*, the set of queries Q_k such that $l_k \le b_{i-1} < b_i \le u_k$, for the region (l_k, u_k) of Q_k. Then,

Definition 5. *Delta query sets, $+DQSet_i$ and $-DQSet_i$*

- $+DQSet_i = QSet_i - QSet_{i-1}$
- $-DQSet_i = QSet_{i-1} - QSet_i$

A query Q_k is determined as an element of $+DQSet_i$ if it covers the region segment N_i, but not N_{i-1}. As the domain space is fully partitioned with the query ranges, Q_k is the query of which the range starts from the lower bound of N_i, i.e., $Q_k \in +DQSet_i$ if $l_k = b_{i-1}$. Likewise, a query Q_k belongs to $-DQSet_j$ if it covers the region segment N_{j-1}, but not N_j. Q_k is the query of which the range ends at the lower bound of N_j, i.e., $Q_k \in -DQSet_j$ if $u_k = b_{j-1}$.

In Fig. 3, 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 $\{Q_3\}$ as a $-DQSet_5$.

4.2 Query Registration and Deregistration

A query can be dynamically registered and deregistered in BMQ-Processor. Assume that a query Q_{in} whose range is (l_{in}, u_{in}) is registered. First, BMQ-Processor 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-Processor 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-Processor 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-Processor 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}.

4.3 Incremental Processing Algorithm

BMQ-Processor incrementally derives differential queries through linear traversals from a previous matching node to a current matching node. The delta queries of the visited nodes are retrieved and then transformed into the differential queries. Due to the locality of data streams, an updated data tuple probably remains in the same node. Even if it does not, it is highly possible that an updated data tuple falls in a nearby node. Therefore, differential queries are quickly derived with a small number of node visits.

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 $b_{h-1} \le v_t < b_h$. Two differential query sets, $QSet^+(t)$ and $QSet^-(t)$ are evaluated while traversing from N_j to N_h as shown in Lemma 1.

Lemma 1.

$$\begin{cases} \text{If } j < h, & QSet^+(t) = [\bigcup_{i=j+1}^{h} +DQSet_i] - [\bigcup_{i=j+1}^{h} -DQSet_i] \\ & QSet^-(t) = [\bigcup_{i=j+1}^{h} -DQSet_i] - [\bigcup_{i=j+1}^{h} +DQSet_i] \\ \text{If } j > h, & QSet^+(t) = [\bigcup_{i=j}^{h+1} -DQSet_i] - [\bigcup_{i=j}^{h+1} +DQSet_i] \\ & QSet^-(t) = [\bigcup_{i=j}^{h+1} +DQSet_i] - [\bigcup_{i=j}^{h+1} -DQSet_i] \\ \text{If } j = h, & QSet^+(t) = QSet^-(t) = \phi. \end{cases}$$

Proof.

By Lemma 2, if $j < h$,

$QSet_h = (QSet_j \cup [\bigcup_{i=j+1}^{h} +DQSet_i]) - [\bigcup_{i=j+1}^{h} -DQSet_i]$ ····· ①

$QSet^+(t) = QSet(t) - QSet(t-1)$ (*by definition 4*)

$= QSet_h - QSet_j$

$= \{(QSet_j \cup [\bigcup_{i=j+1}^{h} +DQSet_i]) - [\bigcup_{i=j+1}^{h} -DQSet_i]\} - QSet_j$ (*by* ①)

$= [\bigcup_{i=j+1}^{h} +DQSet_i] - [\bigcup_{i=j+1}^{h} -DQSet_i]$

$QSet^-(t) = QSet(t-1) - QSet(t)$ (*by definition 4*)

$= QSet_j - QSet_h$

$= QSet_j - \{(QSet_j \cup [\bigcup_{i=j+1}^{h} +DQSet_i]) - [\bigcup_{i=j+1}^{h} -DQSet_i]\}$ (*by* ①)

$= [\bigcup_{i=j+1}^{h} -DQSet_i] - [\bigcup_{i=j+1}^{h} +DQSet_i]$

($\because$ $QSet_j$ and $[\bigcup_{i=j+1}^{h} +DQSet_i]$ are *mutually exclusive*, and $[\bigcup_{i=j+1}^{h} -DQSet_i] \subset (QSet_j \cup [\bigcup_{i=j+1}^{h} +DQSet_i])$)

$\therefore$ The given formula is correct when $j < h$.

The given formula can be proved when $j > h$ as above.
The given formula is trivial when $j = h$.
End of proof

Lemma 2.

Let Q_k, $1 \le k$, be a query with selection region (l_k, u_k). Let $QSet_i$ be the set of queries matching a region segment (b_{i-1}, b_i), i.e., the set of queries Q_k such that $l_k \le b_{i-1} < b_i \le u_k$, for the region (l_k, u_k) of Q_k. Then,

$$QSet_h = \begin{cases} (QSet_j \cup [\bigcup_{i=j+1}^{h} +DQSet_i]) - [\bigcup_{i=j+1}^{h} -DQSet_i], & \text{if } j < h \\ (QSet_j \cup [\bigcup_{i=j}^{h+1} -DQSet_i]) - [\bigcup_{i=j}^{h+1} +DQSet_i], & \text{if } j > h \\ QSet_j, & \text{if } j = h \end{cases}$$

Proof.

Consider the case of $j < h$.

By induction,

1) If $h = j + 1$, then the given formula becomes

$QSet_h = (QSet_j \cup [\bigcup_{i=j+1}^{h} +DQSet_i]) - [\bigcup_{i=j+1}^{h} -DQSet_i]$

$QSet_h = (QSet_j \cup [+DQSet_{j+1}]) - [-DQSet_{j+1}]$

$QSet_h = (QSet_j - [-DQSet_{j+1}]) \cup [+DQSet_{j+1}]$

($\because$ $-DQSet_{j+1}$ and $+DQSet_{j+1}$ are *mutually exclusive)*

$= (QSet_j - [QSet_j - QSet_{j+1}]) \cup [QSet_{j+1} - QSet_j]$ (*by definition 5*)

$= [QSet_j \cap QSet_{j+1}] \cup [QSet_{j+1} - QSet_j]$

$= [QSet_{j+1} \cap QSet_j] \cup [QSet_{j+1} \cap QSet_j^C]$

$= QSet_{j+1} \cap [QSet_j \cup QSet_j^C] = QSet_{j+1} = QSet_h$

$\therefore$ if $h = j + 1$ then the given formula is correct.

2) Let us assume that the given formula is true when $h = k$ (for $k \ge j + 1$), then

$QSet_k = (QSet_j \cup [\bigcup_{i=j+1}^{k} +DQSet_i]) - [\bigcup_{i=j+1}^{k} -DQSet_i]$

$QSet_{k+1} = (QSet_k \cup [+DQSet_{k+1}]) - [-DQSet_{k+1}]$ (*by definition 5*)

$= (QSet_j \cup [\bigcup_{i=j+1}^{k} +DQSet_i] - [\bigcup_{i=j+1}^{k} -DQSet_i] \cup [+DQSet_{k+1}]) - [-DQSet_{k+1}]$

$= (QSet_j \cup [\bigcup_{i=j+1}^{k} +DQSet_i] \cup [+DQSet_{k+1}]) - [\bigcup_{i=j+1}^{k} -DQSet_i] - [-DQSet_{k+1}]$

$= QSet_j \cup [\bigcup_{i=j+1}^{k+1} +DQSet_i] - [\bigcup_{i=j+1}^{k+1} -DQSet_i]$

$\therefore$ The given formula is correct when $h = k + 1$ if it is correct when $h = k$.

$\therefore$ The given formula is correct when $j < h$ by 1) and 2).

The case with $j > h$ can be similarly proved as above.
The case with $j = h$ is trivial.
End of proof

4.4 Analysis of Processing and Storage Costs

The processing cost of BMQ-Processor 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 i^{th} data value and M is the total number of 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 processing cost of BMQ-Processor can be formulated as $\Theta(2 \times N_q \times FL)$.

The storage cost of BMQ-Processor 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 sources, N_d. Consequently, the total storage cost of BMQ-Processor is $\Theta(2N_q + N_d)$.

5 Multi-dimensional BMQ-Processor

For many applications, BMQ processing is also required for multi-dimensional data. For example, location-based services need to query and process two-dimensional or even three-dimensional data. Many sensor networks are composed of multi-functional sensors, which generate multi-attribute sensing values, e.g., both temperature and humidity.

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

Our solution approach is advantageous in several ways. First, it has significantly low storage cost. As in the one-dimensional case, a query is not repeatedly saved in multiple N-dimensional regions, but only twice for each dimension, i.e., $2N$ times total. Note that in other existing approaches such as a grid-based index, a query is repeatedly saved in multiple grids. Second, it has a high processing performance. Our analysis shows that the processing algorithm requires only $(N-1)\sqrt{N}$ times of processing time for one-dimensional BMQ-Processor. For example, two-dimensional processor takes only $\sqrt{2}$ times as much processing time as the one-dimensional processor does. Finally, while multi-dimensional BMQ-Processor is advantageous in its performance and storage cost, it is also simple in its data structures and processing algorithms. We believe that simplicity is an important feature especially from a practical point of view; it can be easily implemented or incorporated in many different systems.

In the rest of this section, we present two-dimensional BMQ-Processor for the ease of explanation.

5.1 Data Structure

Two-dimensional BMQ-Processor consists of following data structures: two RS lists (an RS-X list and RS-Y list), a stream table, and a query table. Fig. 4 shows an example of the processor with three registered queries. The RS-X list is a list of region segments that together comprise the range of the X-dimension, $<RS\text{-}X_1, RS\text{-}X_2, \ldots, RS\text{-}X_n>$. 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.

In the two-dimensional case, each entry of the stream table has two pointers, P_x and P_y, pointing $RS\text{-}X_i$ which contains the current X-dimension value of the stream, and $RS\text{-}Y_i$ which contains the current Y-dimension value of the stream. Also, the current data value is saved for the next processing 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 the borders of queries; it is required for the cross-check algorithm.

5.2 Query Registration and Deregistration

Two-dimensional BMQ-Processor 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 processor, an X-dimension predicate, (x_l, x_u), is registered to the RS-X list and an Y-dimension predicate, (y_l, y_u), is registered to 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.

5.3 Processing Algorithm

Upon an arrival of a data value, two-dimensional BMQ-Processor computes $QSet^+$ and $QSet^-$. Fig. 5 shows overall flow of the processing algorithm. The first step of the algorithm is to calculate differential query sets for each dimension: $\pm XQSet$ and $\pm YQSet$. This is simply done by applying one-dimensional incremental processing algorithm to the RS-X list and RS-Y 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})

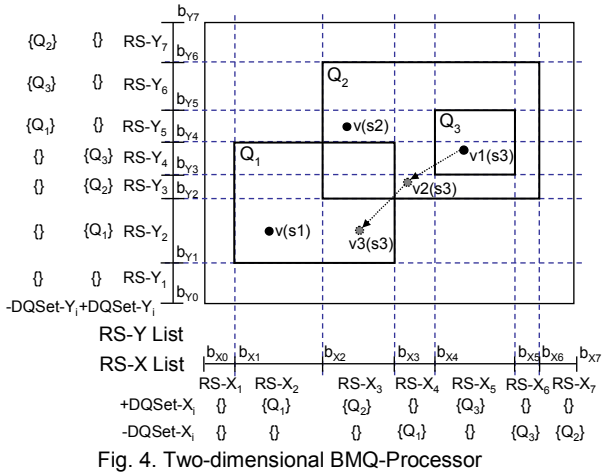


Fig. 4. Two-dimensional BMQ-Processor

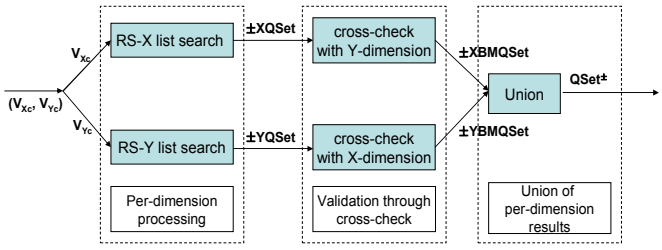


Fig. 5. Flow of a processing algorithm

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

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

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

Obtain si's previous data value, (vXp, vYp), using the stream table
For each element Qi of –XQSet
    Get Qi's Y-dimension predicate, (Qi_yl, Qi_yh), from the query table
    If(Qi_yl <vYp< Qi_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 */
```

Fig. 6. Cross-check algorithm

The second step is to validate if crossing of either the X- or Y-dimensional border really results in crossing of a two-dimensional border. We developed an efficient cross-check algorithm described in Fig. 6. The algorithm examines the borders of the unchecked dimensions of the queries in per-dimension differential query sets.

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.

5.4 Analysis of Processing and Storage Costs

The processing performance of multi-dimensional BMQ-Processor 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-Processor 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 processing cost is $(d-1)\sqrt{d}$ times as much as that of one-dimensional BMQ-Processor, thereby being $O((d-1)\sqrt{d} \times 2N_q \times FL)$, where N_q is the number of BMQs and FL is multi-dimensional fluctuation level (see details in Appendix **B**).

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-Processor. Additionally, multi-dimensional BMQ-Processor maintains a query table, thus the storage cost of multi-dimensional BMQ-Processor is $\Theta(d(2N_q + N_d) + N_q)$, where N_d is the number of input data sources.

6 Region Transition Monitoring

The semantics of border monitoring is also important as it forms the basis for various, more complex and advanced monitoring. As a sample case of such monitoring, we develop *region transition monitoring*. The focus of the extension lies in monitoring a pair of consecutive border crossings beyond an individual crossing. Specifically, it is monitored whether the consecutive border crossings occur or not within a given time. This type of monitoring is practically important because it represents *transition or stay of a stream associated with a region*. We first describe region transition monitoring with a service scenario, and then extend BMQ-Processor to support it.

6.1 Example Scenario

In a city, there are a plenty of parking garages. For users' convenience, each parking garage tries to run an advanced toll system. The system waives tolls for the cars which stay in the parking lot only for a short time. Some cars come into the garage and go out, e.g., within three minutes. Also, it re-validates formerly paid tolls for the cars which have paid their tolls and temporarily left the garage. Some may come back shortly, e.g., in three minutes, and would like to re-enter the garage but without paying tolls again. For these purposes, it is necessary to monitor the occurrence of paired border-crossing events, i.e., the transition of cars associated with the parking lot.

On the other hand, the system charges tolls for the cars which come into the garage and stay for more than a given time. Also, it invalidates formerly paid tolls for the cars which left the garage and stay outside for more than a given time. For these purposes, it is necessary to monitor the non-occurrence of paired border-crossing events, i.e., the stay of cars associated with the parking garage.

6.2 Definition of RTMQ

To facilitate the region transition monitoring, we define a Region Transition Monitoring Query (RTMQ). An RTMQ is represented as *RTMQ*(*r*, *t*), where

- *r* is a region of interest
- *t* is a time constraint

Given *r* and *t*, an RTMQ specifies four types of *region transition events* (RTEs) for all input data streams. The four types of RTEs are as following.

Type	Semantics
I ▷ O	Transition such that a stream comes into and then goes out of a region *r* within time *t*
O ▷ I	Transition such that a stream goes out of and then comes into a region *r* within time *t*
I ▷ ¬O	Stay such that a stream comes into and then does not go out of a region *r* within time *t*
O ▷ ¬I	Stay such that a stream goes out of and then does not come into a region *r* within time *t*

* ▷ means that the right event occurs after the left one.
* ¬ means negation.

The two parameters r and t specify filtering conditions: (1) r is the interest range for the generation of I and O events (2) t constrains the time window, i.e., only I and O events within the window t are valid. In addition, an equality condition on stream_id is implied, i.e., sequenced I and O events should refer to the same stream sources.

In the context of event-based systems, RTEs can be considered as special types of composite events. Specifically, **O⊳I** and **I⊳O** are specified as a binary sequence, i.e., a SEQ operator in [11][33][58][64], of I and O events with a time window t. **O⊳¬I** and **I⊳¬O** can similarly be specified additionally with negation operators. An RTMQ collectively specifies all RTEs from a large number of input streams and is especially useful in many location-based applications.

6.3 Approach for Efficient RTMQ Processing

As in processing BMQs, RTMQ evaluation involves massive event processing over a large number of data streams. Accordingly, we should carefully consider performance issues for RTMQ processing. In the following, we present our approach and discuss alternative approaches.

For RTMQ processing, we first need to detect I and O events. Then post-processing is required (1) to compose only I and O events from the same source and (2) to inspect if the matching I and O events satisfy the time constraint t. It is more important to efficiently handle the first round, i.e., the detection of I and O events, since it directly operates on the high-rate input data streams. The post-processing operates on the derived I and O event streams with much lower rates. As BMQ-Processor is adept at processing the first round, it makes the whole processing efficient.

We develop an efficient algorithm, *Transition-Sequence* (Tran-Seq), for the post-processing, which requires SEQ operation along with a time constraint. (The algorithm is described in Section 6.4.) Tran-Seq uses sliding widows as well as hash tables for fast matching of event streams. In this sense, it is similar to the *sliding window hash join* algorithm, which is recognized as the state-of-the-art for the continuous evaluation of equi-join operations [25][37]. Given an *RTMQ*(r, t), the four types of RTEs are processed by sharing the same windows and hash tables. Also, I and O events from multiple sources share the same structures.

Existing methods for composite event detection can be used for the post-processing [11][33][58]. In general, to handle aggregated event streams from multiple sources, they first de-multiplex the event streams based on stream_id. Then, they evaluate the composition of I and O events for each stream source along with checking a time constraint. Compared to these approaches, the proposed one is advantageous in that it processes I and O events from multiple sources using the same Tran-Seq.

Recently, SASE was proposed as an efficient method for the detection of composite events from large windows [64]. Similar to Tran-Seq, it processes aggregated event streams through a single NFA data structure. It also supports efficient sliding window-based filtering as well as equality testing on multi-source event streams. While SASE is very efficient and comparable to Tran-Seq, the latter is a better choice for the post-processing of RTMQs. Using SASE, given an *RTMQ*(r, t), the four different types of RTEs should separately be processed with separate data structures, i.e., a separate NFA, sliding window, and stack for intermediary results. Our inspection shows that SASE requires at most four times more in processing and storage costs than the proposed method.

6.4 Processing Algorithm and Cost Analysis

Fig. 7 shows the processing flow of *RTMQ*(r, t). At first, I and O events with respect to a *BMQ* with range r are derived using BMQ-Processor. Then, *Tran-Seq*(t) composes them while checking equality on stream_ids and time constraint. Importantly, **O⊳I** and **I⊳¬O** are derived by I⋈O, meaning searching the O window to match an incoming I event, and **I⊳O** and **O⊳¬I** by I⋉O, searching for the I window to match a new O event. Since the two operations are symmetrically processed, we explain only I⋉O for conciseness. The I⋉O operation sequentially executes three sub-operations: 1) insert, 2) probe, and 3) delete. The probe operation derives **I⊳O**, and the delete extracts **O⊳¬I**. Fig. 8 shows the details of the operations. Note that the probe operation includes the removal of the matched event. For example, without 2-3) in Fig. 8, the matched I event (**I⊳O**) would later be included as **I⊳¬O**. It is a wrong result since a stream already transited through the range is recognized as a staying stream.

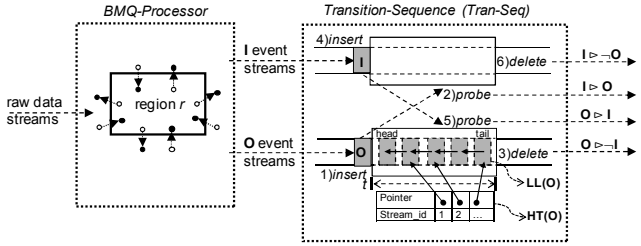


Fig. 7. Processing flow of RTMQ

Input: an O event (derived from BMQ-Processor)
Output: **I⊳O** and **O⊳¬I**

Four data structures: LL(I), HT(I), LL(O), HT(O)
(* Linked List (LL) and Hash Table(HT) for I and O event streams.
LL is used for buffering and sliding events within *t*.
HT is used for probing events by a stream_id.)

1) *insert*:
  1-1) Insert the event into the head of LL(O) with timestamp
  1-2) Insert the stream_id of the event into HT(O) with pointer

2) *probe*:
  2-1) Probe HT(I) by using the stream_id of the event
    If (there exist an I event which has the same stream_id) {
      2-2) Output the matched result as **I⊳O**
      2-3) Delete the matched I event from LL(I) and HT(I)
    }

3) *delete*:
  3-1) Identify the O events which exceed the temporal constraint by traversing from the tail of LL(O)
  3-2) Output the identified O events as **O⊳¬I**
  3-3) Delete the identified O events from LL(O) and HT(O)

Fig. 8. Detailed operations of I⋉O

Notation	Meaning
σ_{BMQ}	selectivity of a BMQ
λ_{ds}	arrival rate of raw data streams
λ_I	arrival rate of I events
λ_O	arrival rate of O events
λ_e	arrival rate of both I and O events
$\|B\|$	number of hash buckets
σ_w	selectivity factor of I and O window

BMQ filtering	$\lambda_{ds} \times \sigma_{BMQ} = \lambda_I + \lambda_O = \lambda_e$
Costs of a Tran-Seq	Processing cost $= \lambda_I\{$ *insert*(I) + *probe*(O) + *delete*(I)$\}$ $+ \lambda_O\{$*insert*(O) + *probe*(I) + *delete*(O)$\}$ $\simeq \lambda_e\{2 + (t\lambda_e/(2\cdot\|B\|) + 2\cdot\sigma_w) + 2\}$ Storage cost = Two hash tables and two linked lists $= 2t\lambda_I + 2t\lambda_O = 2t\lambda_e$ (* We assume that I and O events show similar occurrence frequency since they are symmetric. Thus, $\lambda_I \simeq \lambda_O \simeq \lambda_e/2$)

Fig. 9. Analytical costs for a Tran-Seq

The cost of RTMQ processing is determined by BMQ-Processor and Tran-Seq. Since the costs for BMQ-Processor are described Section 4.4 and 5.4, we only analyze the costs for Tran-Seq (see Fig. 9). The processing cost for Tran-Seq depends on the rate of I and O events and the cost of operations performed upon the arrival of an event. First, the rate of the events, λ_e is represented as the multiplication of the rate of raw data streams, λ_{ds} and the selectivity of BMQ, σ_{BMQ}. Second, the costs of Tran-Seq operations are as follows. Upon the arrival of an event, insert and delete operations require two accesses[2] (i.e., one access to the linked list and one access to the hash table), respectively. An event is inserted into a hash bucket in the order of arrivals and deleted in the reverse order. The cost of the probe operation depends on both the window size, $t\lambda_e/2$ and the number of hash buckets, $|B|$. When probing the hash table, $t\lambda_e/(2\cdot|B|)$ tuples must be accessed in the worst case, which is the number of events in a hash bucket. If there is a matched event, corresponding deletions on the linked list and hash table follow and require two accesses, i.e., $2\cdot\sigma_w$. Finally, the storage cost is decided by the two hash tables and two linked lists.

In addition, we estimate the real costs through an experiment based on the previous parking garage scenario. We realistically model and simulate the location updates of a million vehicles in a part of Seoul city (i.e., location updates every 30 sec). Each garage is modeled as an RTMQ with a range of size 30m x 30m and time constraint of 3 minutes. Through the experiment, we measured the three parameters: λ_{ds}, λ_e, and σ_w. They are 33,300 data/sec, 21 events/sec and 0.5, respectively. We observe that the I and O event rate, λ_e, is significantly reduced through BMQ filtering compared to the raw data rate, λ_{ds}. It is trivial to see that processing over low-rate event streams is always better than processing over high-rate raw data streams. The impact of efficient filtering of BMQ-Processor will increase with multiple RTMQs.

[2] For concise analysis, we regard the cost of a single operation on LL or HT as one access, i.e., the unit of cost, as in [37].

The cost of Tran-Seq processing is estimated from the above measured values, i.e., $\lambda_e\{2+(t\lambda_e/(2\cdot|B|)+2\cdot\sigma_w) + 2\} = 126$ accesses/sec. In the estimation, we set $|B|$ to the same number as the window size, i.e., $t\lambda_e/2 = 1{,}890$. A hash table of this size is more than viable for a typical in-memory hash table implementation. The cost, i.e., 126 accesses/sec, is almost negligible compared to the raw input data access cost, i.e., λ_{ds} = 33,300 accesses/sec. Through these analyses, we see that our mechanism consisting of BMQ-Processor and Tran-Seq efficiently processes RTMQs.

7 EXPERIMENTS

In this section, we discuss the results of our performance study on BMQ-Processor. The experiment consists of three parts: performance study on one-dimensional BMQ processing, that on two-dimensional BMQ processing, and that on multi-dimensional BMQ processing. We compare the processing performance and storage cost of BMQ-Processor with a RMQ evaluation-based mechanism, namely *DiffRMQ*. DiffRMQ derives differential query sets in two steps. It first retrieves matching query sets for consecutive data values. Then, it performs a set difference operation on the resulting matching query sets of two consecutive data values and removes the queries containing both data values. The state-of-the-art RMQ evaluation methods are used for DiffRMQ; CEI [63] and IS-list [27] for one-dimensional BMQ processing and CES [62] for two-dimensional BMQ processing. The experiments are conducted using a machine equipped with P-III 1GHz CPU, 512MB RAM, and Linux 2.4.

7.1 Performance of One-dimensional BMQ Processing

7.1.1 Experimental Setup

Stream generation. For this experiment, we consider the financial trading scenario described in Section 3.1. Based on the observation on Korean stock market (see Appendix **A**), we synthetically generate stock price streams. We vary FL from 0.01% to 0.1%, as observed from the traces. We use 2000 stream sources as the input, and each stream contains 1,000 data tuples. Initial stock prices follow uniform distribution.

Query generation. Queries specify the ranges of stock prices to be monitored. We distribute the queries by locating lower bounds of query ranges between 1 and D – 1, where D is fixed to 1,000,000, the maximum price of most Korean stocks [38]. The lower bounds follow a uniform distribution. In practice, the width of a query, W, is usually larger than FL. We set W to 1 ~ 10 times larger than FL. W is also normalized with respect to the domain size. Finally, the number of queries, N, varies from 10K to 100K.

7.1.2 Processing Performance

In this experiment, we compare the performance of BMQ-Processor with that of DiffRMQ. For intuitive comparison, we first measure the *Processing Efficiency* (*PE*) which is defined as follows.

$$PE = \frac{\sum_{i=1}^{M} \text{size of final result}}{\sum_{i=1}^{M} \text{size of intermediate result}} \times 100(\%)$$

(M is the total number of input data tuples)

In BMQ-Processor, the size of the intermediate result is the total number of retrieved delta queries during an evaluation. In DiffRMQ, it is the number of the matching queries. In both cases, the size of the final result is the total number of differential queries after evaluation. We then measure the average processing time, which is measured as the elapsed time to produce the final result per data tuple. We measure the PE and the average processing time while varying three parameters: the number of queries N, the width of queries W, and the fluctuation level FL.

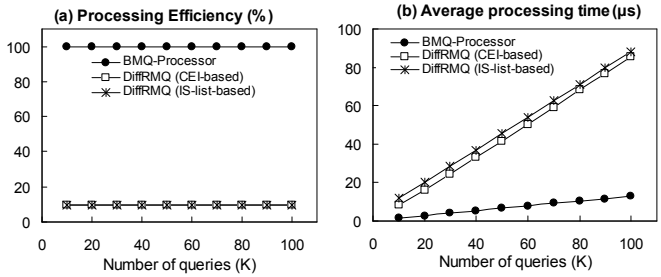


Fig. 10. Effect of the number of queries (W=0.1%, FL=0.01%)

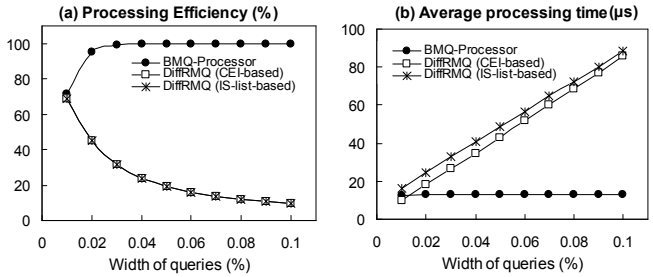


Fig. 11. Effect of the width of queries (N=100K, FL=0.01%)

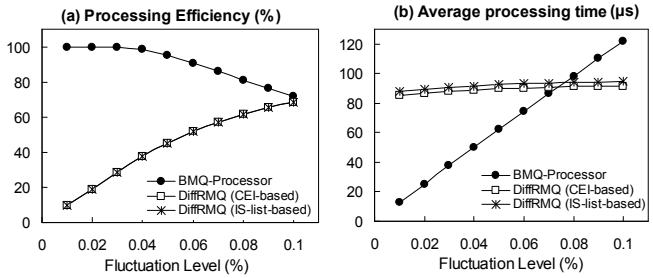


Fig. 12. Effect of the Fluctuation Level (N=100K, W=0.1%)

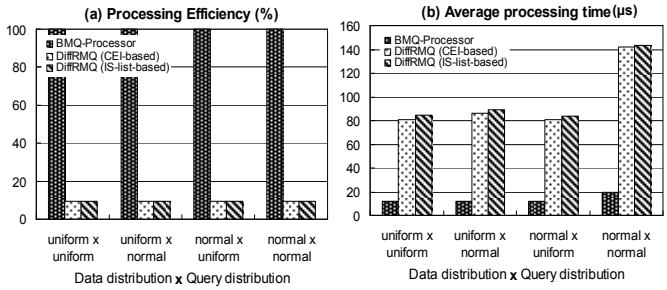


Fig. 13. Effect of the distribution (N=100K, W=0.1%, FL=0.01%)

First, we vary N from 10K to 100K. W and FL are fixed to 0.1% and 0.01%, respectively. Fig. 10 shows the PE and the average processing time as a function of N. The PE of BMQ-Processor is 100% regardless of the number of queries and much higher than the PE of DiffRMQ. In DiffRMQ, consecutive matching query sets are likely to much overlap due to the locality in data streams. Thus, many irrelevant queries are accessed to obtain only a few differential queries. Consequently, the processing time of DiffRMQ is significantly longer than that of BMQ-Processor. The slight increase in the processing time of BMQ-Processor mainly comes from the increase in the final result size, which tends to grow with the number of registered queries.

Second, we vary W from 0.01% to 0.1%. N and FL are fixed to 100K and 0.01%, respectively. Fig. 11 shows the results. As the width of the queries increases, the PE of DiffRMQ rapidly decreases, whereas that of BMQ-Processor remains almost 100%. It is because the size of the intermediate result of DiffRMQ increases with the width of queries. However, that of BMQ-Processor is not affected by the width of queries. The size of final result does not change in neither of the cases. Therefore, as the width of queries increases, the processing time of DiffRMQ increases significantly, but that of BMQ-Processor remains constant.

Third, we vary FL from 0.01% to 0.1%. N and W are fixed to 100K and 0.1%, respectively. As FL increases, the size of the intermediate result of BMQ-Processor increases while that of DiffRMQ changes little. The size of the final result increases in both cases. Thus, the PE of BMQ-Processor decreases slowly while that of DiffRMQ increases rapidly as shown in Fig. 12 (a).

Interestingly, the PE of BMQ-Processor is higher than that of DiffRMQ as long as FL is smaller than W. It is because the average intermediate result size of DiffRMQ is $2W \cdot N^3$ while that of BMQ-Processor is $2FL \cdot N$. Accordingly, the processing time of BMQ-Processor is smaller than that of DiffRMQ if FL is smaller than W. If FL increases up to W, the processing time of BMQ-Processor becomes slightly larger than that of DiffRMQ due to the increasing cost of RS node traversals. However, W is generally larger than FL in practical cases.

To investigate the effect of skewed data and query dis-

[3] Given a data value, the number of matching queries is $W \cdot N$ in RMQ processing method. In DiffRMQ, two sets of matching queries must be accessed for previous and current values to perform a set difference operation.

tributions, we perform an additional experiment. We use a normal distribution to simulate the skewed distributions of data and queries. The distributions for data and queries have the same mean value, i.e., D/2. We generate four different combinations of data and query distributions to evaluate the performance under various situations. N, W and FL are fixed to 100K, 0.1% and 0.01%, respectively.

The change of the distributions does not affect the PE as shown in Fig. 13(a). This means that the ratio of the intermediate result size over the final result size is the same regardless of the distributions. The PE of BMQ-Processor remains 100%, which is ten times larger than that of DiffRMQ.

As shown in Fig. 13(b), when either one of the data and query distributions is skewed, the processing time changes little. Consider the case that only queries are skewed. In the regions where the query borders are densely located, the processing time for a corresponding data value increases. However, it decreases in sparse regions. As a result, the average processing time does not change. When only the data distribution is skewed, the average processing time is not affected.

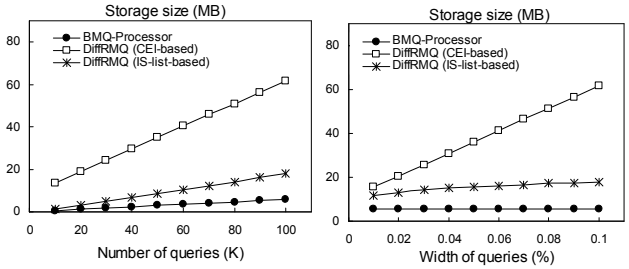
An interesting result is that the average processing time is considerably larger when both data and query distributions are skewed. In this case, many of the values in the streams are located in the region where the query borders are densely located. Thus, both the final and intermediate result sizes increase, resulting in an increase in the processing time. Since the intermediate result size of BMQ-Processor is much smaller than that of DiffRMQ, BMQ-Processor is more efficient in terms of processing time. In conclusion, BMQ-Processor is more robust against skewed data and query distributions than DiffRMQ.

7.1.3 Storage Cost

In this experiment, we compare the storage cost of BMQ-Processor to that of DiffRMQ. We measure the size of the memory space used for each processing method. To identify the effect of N and W on storage cost, we run two experiments. In the first experiment, we vary N from 10K to 100K and fix W to 0.1%. In the second, we vary W from 0.01% to 0.1% and fix N to 100K.

Fig. 14(a) shows the result of the first experiment. BMQ-Processor uses much less memory than the two DiffRMQs, especially than CEI-based DiffRMQ. In general, RMQ processing methods store queries redundantly. CEI stores a query into multiple grids covered by its range. Even a tree-based method, i.e., IS-list, stores a query log N times (N is the number of registered queries). In contrast, BMQ-Processor stores a query only twice, thereby resulting in considerably smaller storage cost.

Fig. 14(b) shows the storage size as a function of query width. The storage cost of CEI-based DiffRMQ increases rapidly with query width. In CEI, a query with a wide range is repeatedly inserted to the grids overlapping the range, resulting in high storage cost. As for IS-list-based DiffRMQ, its storage cost is less affected by query width as it is a tree-based approach. BMQ-Processor shows a constant storage usage regardless of query width since it only stores two delta queries upon each query registration.

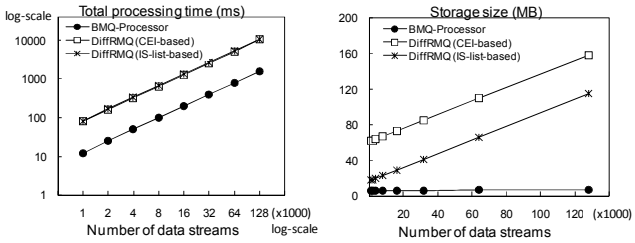


(a) Effect of N (W=0.1%) (b) Effect of W (N=100K)

Fig. 14. Storage cost

7.1.4 Scalability with the Number of Data Streams

We test the scalability of BMQ-Processor as increasing the number of data streams, *S*, from 1K to 128K in log-scale. This experiment is important since BMQs monitor a large number of data streams in our target application scenarios. For the experiment, N, W and FL are fixed to 100K, 0.1%, and 0.01%, respectively. We measure the storage cost and total processing time taken for processing a data tuple from each data stream, i.e., *S* tuples in total.



(a) Processing cost (b) Storage cost

Fig. 15. Scalability with the number of data streams

As shown in Fig. 15(a), total processing time of BMQ-Processor grows linearly with the number of data streams in log-log scale. This linear growth is also shown in the DiffRMQs' cases. It is shown that the total processing time of BMQ-Processor is 6.8 times smaller than those of DiffRMQs. The time gap does not change with varying number of data streams. The significant gap in processing times comes from the similar reasons discussed in 7.1.2. The result conforms to the observation that the processing time additively increases with additional streams to the input. It is because data tuples from different data streams are separately processed without intervening each other. As mentioned, the processing time per tuple does not change when N, W, and FL are fixed. Fig. 15(b) shows storage costs. The storage consumption of BMQ-Processor is the lowest, and hardly increases. It is because BMQ-Processor only adds a stream_id and a node_pointer for a stream source, which requires a negligible amount of memory space. In contrast, DiffRMQs require maintaining previous matching query set per stream source. This results in linear increases of the storage usage as the number of streams increases.

7.2 Performance of Two-dimensional BMQ Processing

7.2.1 Experimental Setup

Stream generation. For this experiment, we consider the location-based advertisement scenario introduced in Section 3.1. We use Network-based Generator of Moving Objects (NGMO[49]) to generate location data streams. The road map of Oldenburg, a city in Germany is used for the data generation. The map is 15km by 15km in size. To generate streams with different FL, we assume that there are two kinds of moving objects: fast objects such as vehicles and slow objects such as pedestrians. The maximum speed of the fast objects is 60km/h and the average is 20km/h. Similarly, the slow objects have 12km/h of maximum speed and 4.3km/h of average speed. The number of the moving objects, i.e., stream sources, is set to 1,000, and the location update period is 30 sec. We collect 100 location data from each stream source.

Query generation. Queries specify advertisement regions, which are monitored to identify border-crossings. We assume that all advertisement regions are square. Stores are usually located densely in downtown area of a city, and hence, a query distribution is likely skewed. Thus, we use a normal distribution with the mean value being the center of the city. We expect that the number of advertising stores in a city is up to several thousands, and vary the number of queries from 0.5K to 5K. Different stores advertise in the regions of different sizes. For example, a gas station may send its electronic coupons within a several-kilometer region, while a restaurant, within a several-hundred-meter region. We vary the width of queries from 0.5km to 5km.

7.2.2 Processing Performance

First, we vary N from 0.5K to 5K. W is fixed to 5km. Fig. 16 shows the PE and the average processing time as a function of the number of queries. Compared to the one-dimensional case, the PE of BMQ-Processor is smaller because the size of the intermediate result increases but many of them are not included in the final result. However, the PE of BMQ-Processor is still much higher than that of CES-based DiffRMQ. Accordingly, the processing time of BMQ-Processor is significantly lower than that of CES-based DiffRMQ with the same FL. The processing time of CES-based DiffRMQ drastically increases as the number of queries increases. It is because more matching queries should be accessed with more registered queries. The processing time of BMQ-Processor also increases slightly. However, the increase is much slower because the number of accessed queries is much smaller than in CES-based DiffRMQ. We can see the effect of FL on the performance of BMQ-Processor in Fig. 16 (b). A fast moving object is likely to incur a larger size of the intermediate and the final results than a slow moving object. As a result, the processing time for a fast moving object is larger than that for a slow moving object. The effect becomes more apparent as the number of queries increases.

Second, we vary W from 0.5km to 5km. N is fixed to 5K. Fig. 17 shows the PE and the average processing time as a function of query width. The overall PE of BMQ-Processor is superior to that of CES-based DiffRMQ. Consequently, the processing time of BMQ-Processor is much lower than that of CES-based DiffRMQ with the same FL. For BMQ-Processor, the processing time is almost constant regardless of query width. On the other hand, that of CES-based DiffRMQ rapidly increases, possibly leading to a critical performance problem. We can also see the effect of the FL on the performance of BMQ-Processor in Fig. 17 (b). The processing time is larger for a fast moving object as explained before.

7.2.3 Storage Cost

We also compare the storage cost of BMQ-Processor with that of DiffRMQ. We present the utilized storage size as a function of N and W. First, we vary N from 0.5K to 5K and fix W to 5km. Second, we vary W from 0.5km to 5km and fix N to 5K.

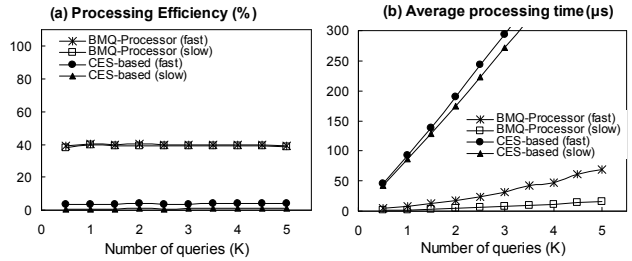


Fig. 16. Effect of the number of queries (W=5km)

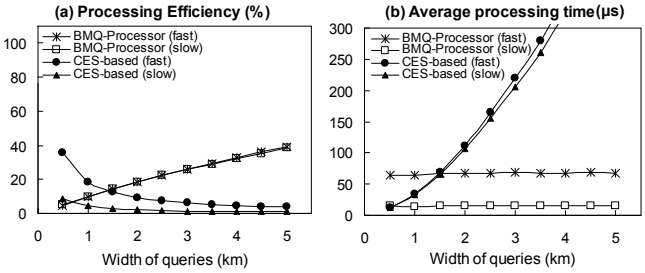
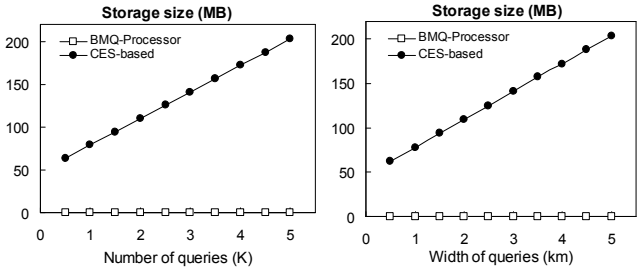


Fig. 17. Effect of the width of queries (N=5K)



(a) Effect of N (W=5Km) (b) Effect of W (N=5K)

Fig. 18. Storage cost

Fig. 18(a) shows the storage size as a function of the number of queries. The storage size for CES-based DiffRMQ increases from 63.5MB to 203MB. Similar to the case of CEI, a query is redundantly stored in multiple grids covered by a query region in CES, resulting in significant storage cost. However, BMQ-Processor consumes much smaller storage space, i.e., less than 1MB since a query is stored only twice for each dimension.

Fig. 18(b) shows the storage size as a function of query width. BMQ-Processor consumes almost a constant amount of storage space, i.e., about 0.38MB, regardless of the width of queries. However, the storage size for CES-based DiffRMQ considerably increases with the width of queries. With an increase in W, i.e., an increase in a query coverage,

the query needs to be stored in more grids of CES, which results in high storage consumption.

In Fig. 18, it is shown that BMQ-Processor consumes at most hundred times smaller storage space than CES-based DiffRMQ. Also, Fig. 14 shows that it consumes at most ten times smaller storage space than CEI-based DiffRMQ. In conclusion, BMQ-Processor is very efficient in storage usage in both two-dimensional and one-dimensional case.

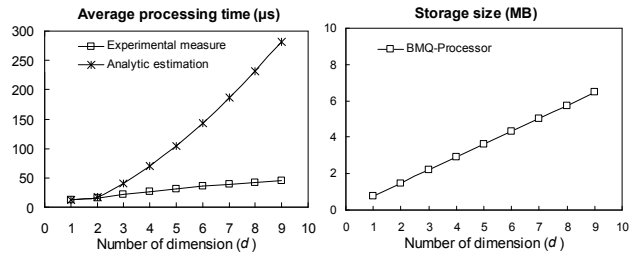
7.3 Performance of Multi-dimensional BMQ Processing

7.3.1 Experimental Setup

This experiment examines the effect of dimensionality on the performance of BMQ-Processor. We assume that each dimension has the same maximum size, i.e., 10^5. The FL of data streams for each dimension is 0.1%. Queries have the same width for every dimension, i.e., 0.1%. We vary the number of dimensions from one to nine. The number of queries is fixed to 10K. We use uniform distribution for data and query generations.

7.3.2 Processing Performance

Fig. 19(a) shows the average processing time as a function of the number of dimensions, *d*. The average processing time increases very slowly with dimension. The figure also shows estimated processing times for comparison. From the analysis described in Section 5.4, the processing time is $(d-1)\times\sqrt{d}\times\tau_1$, where τ_1 is the processing time for one-dimensional case. The estimation has been made using the measured average processing time for one dimensional case. As shown in the figure, the average processing time grows very slowly, compared to the estimated values. It is because BMQ-Processor does not have to perform cross-checking completely with every dimension. Assume that the X-dimensional border of a query has been crossed by a recent data value. Then, it should be validated if the data value really crossed the borders of other dimensions, e.g., y- and z-dimension, as well. However, once it is found that y-dimensional border has not been crossed, the inter-dimensional validation process does not have to continue with z-dimension. In conclusion, the proposed BMQ-Processor scales very well with dimension, and is a practical solution for multi-dimensional BMQ processing.



(a) Processing performance (b) Storage cost

Fig. 19. Effect of the dimensionality

7.3.3 Storage Cost

Fig. 19(b) shows the storage size as a function of the number of dimensions. Because a separate RS list is used for each dimension, the consumed storage size increases linearly to the number of dimensions. Even for 9 dimensional case, BMQ-Processor consumes less than 7MB of memory space. This low storage cost is a great advantage of BMQ-Processor as in-memory processing is important for stream processing.

8 Conclusion

In this paper, we have presented BMQ-Processor, a high-performance border-crossing event detection framework for large-scale monitoring applications. Our study is the first attempt to propose BMQ-Processor which handles a large number of BCEs over numerous high-rate data streams. For this purpose, we develop a novel shared and incremental processing mechanism. For shared processing, BMQ-Processor adopts a query indexing approach, thereby achieving a high level of scalability. For incremental processing, BMQ-Processor utilizes the locality of data streams and accordingly develops a stateful query index. Thus, successive BMQ evaluations are significantly accelerated. Based on the main idea, we design a one-dimensional as well as a multi-dimensional BMQ-Processor to support various monitoring applications. We also discuss region transition monitoring as an attempt to extend border monitoring semantics to more advanced ones. Our extensive experimental study and analysis demonstrate excellent processing performance and low storage cost of BMQ-Processor; BMQ-Processor outperforms the state-of-the-art query index-based evaluation mechanisms by orders of magnitude.

Acknowledgment

We thank the anonymous reviewers for their valuable comments which help significantly improve the quality of the paper.

Appendix

A. Locality of Data Streams

We expect that data streams change gradually in many practical situations. In this appendix, we examine the locality of data streams with real data traces. (The discussion here does not aim at generalizing the existence of locality; such generalization should not be made in haste and may not even be possible. However, we think there are also many practical cases showing locality.) We first analyze two numerical data streams: the stock price of Samsung and LG Electronics [38]. Fig. 20 shows their prices from April 14th to April 22nd, 2004. As we expected, the stock prices change gradually. Especially, LG's stock price changes more gradually than Samsung's.

In order to quantify the degree of locality of a given data stream, we defined *Fluctuation Level* (FL) in Section 4.4. Note that the degree of locality is high when differences between data values are small, and vice versa. That is, *the degree of locality is inversely proportional to FL*.

In order to identify FL's dependence on data sources as well as the sampling rate, we plot FL for the each data source according to the sampling rate. As shown in Fig. 21, the FL of the LG stock data is smaller than that of Samsung's. Thus, the degree of locality for LG's stock data is higher than that of Samsung's. More important,

the FL decreases as the sampling rate increases in both data streams, which means that the degree of locality increases.

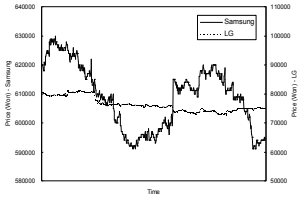


Fig. 20. Real stock prices

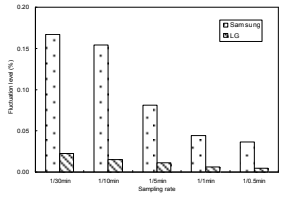
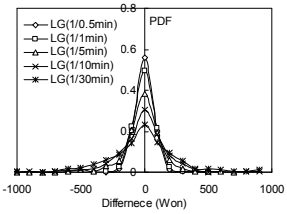


Fig. 21. FL of stock prices



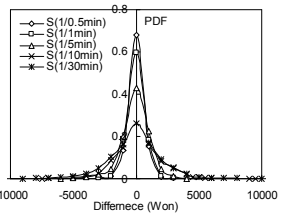


Fig. 22. PDF for difference of two consecutive values in stock prices

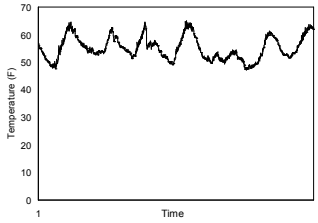


Fig. 23. Real temperature

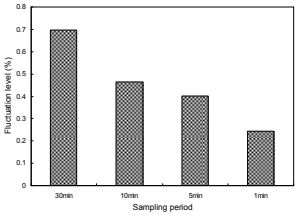


Fig. 24. FL of temperature

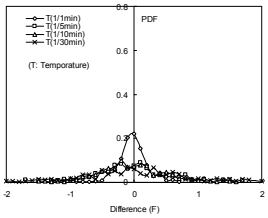


Fig. 25. PDF of difference

Although this is as expected, we note that it has an interesting implication. Usually, increasing the update frequency directly increases the processing load, resulting in severe performance problems. However, BMQ-Processor does not incur much processing cost even at high data rates. It is mainly because the degree of locality increases according to the increase in the sampling rate.

Finally, Fig. 22 shows the probability density function (PDF) of value differences between two consecutive data. They present the characteristics of locality in detail. Even though the shape of distributions is a little bit different from each other, they follow a skewed-distribution centered to zero. As the sampling rate decreases, the distribution becomes more widely spread to the larger difference values. Interestingly, the probability of zero difference, i.e., consecutive values are the same, is considerably high.

We also investigate the sensor data stream, atmospheric temperature data from the University of Washington [57]. Fig. 23 shows temperature data during eight days from May 6th to May 13th 2004. We observed the similar behavior in temperature data as shown in Fig. 24 and 25.

B. Performance analysis of BMQ-Processor

Lemma. *The time complexity of d-dimensional processing operation is* $O(d\sqrt{d}(2N_qFL))$

Proof.

Let $v=(v_1,v_2,v_3,\cdots,v_d)$ be a movement vector from previous data value at *t*-1 to the data value at *t*. Then, the total number of queries retrieved from RS node traversals for *d* dimensions at time *t* is

$2N_q\times\sum_{i=1}^{d}\frac{v_i}{D_i}$ (when queries are uniformly distributed)

The average number of the retrieved queries for *m* times of data updates is

$$\sum_{t=2}^{m}(2N_q\times\sum_{i=1}^{d}\frac{v_i}{D_i})\times\frac{1}{m-1}=2N_q\times\sum_{i=1}^{d}\sum_{t=2}^{m}(\frac{v_i}{D_i}\times\frac{1}{m-1})=2N_q\times\sum_{i=1}^{d}FL_i$$

Let $x\in R^d$, $y\in R^d$ and $x=(FL_1,FL_2,FL_3,\cdots,FL_d)$, $y=(1,1,1,\cdots,1)$

By Cauchy-Schwarz inequality

$$|x\cdot y|=|x_1y_1+x_2y_2+\cdots+x_dy_d|\le\sqrt{x_1^2+x_2^2\cdots+x_d^2}\sqrt{y_1^2+y_2^2\cdots+y_d^2}$$

$$|FL_1+FL_2+\cdots+FL_d|\le\sqrt{FL_1^2+FL_2^2+\cdots+FL_d^2}\sqrt{1^2+1^2+\cdots+1^2}$$

$|FL_1+FL_2+\cdots+FL_d|\le FL\sqrt{d}$ ($\because$ by definition of multi-dimensional FL, $FL_1^2+FL_2^2+\cdots+FL_d^2=FL^2$)

Thus, the total number of queries retrieved from RS node traversals, $2N_q\times\sum_{i=1}^{}FL_i$, can be bounded by $2N_q\times\sqrt{d}FL$.

Since a query retrieved from an RS node traversal should be crossed-checked with other (*d*-1) dimensions, the total processing cost is $O((d-1)\times2N_q\times\sqrt{d}FL)$, which is equal to $O(d\sqrt{d}(2N_qFL))$.

Cf) Cauchy-Schwarz inequality

http://planetmath.org/encyclopedia/CauchySchwarzInequality.html

REFERENCES

[1] D. J. Abadi, S. Madden, and W. Lindner, "REED: Robust, Efficient Filtering and Event Detection in Sensor Networks", *Proc. VLDB*, 2005.

[2] D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul and S. Zdonik, "Aurora: A New Model and Architecture for Data Stream Management". *VLDB Journal*, vol. 12, no. 2, pp. 120-139, August 2003.

[3] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "A Survey on Sensor Networks" *IEEE Communication Magazine*, vol.40, no.8, pp.102-114, August 2002.

[4] A. Arasu, S. Babu and J. Widom, "The CQL Continuous Query Language: Semantic Foundations and Query Execution" *VLDB Journal*, vol. 15, no. 2, pp. 121-142, June 2006.

[5] G. Banavar, M. Kaplan, K. Shaw, R.E. Strom, D.C. Sturman, and W. Tao., "Information Flow Based Event Distribution Middleware", *Proc. ICDCS Workshop*, 1999.

[6] L. Bao and S. S. Intille, "Activity recognition from user-annotated acceleration data", *Proc. Pervasive* 2004.

[7] I. Biswas, "A Composite Event Definition Language and Detection System for the Integration Rules Environment", *Master thesis*, *Arizona state university*, May 2005.

[8] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and Evaluation of a Wide-Area Event Notification Service", *ACM Transactions on Computer Systems*, vol. 19, no. 3, pp. 332-383, 2001.

[9] S. Chakravarthy and R. Adaikkalavan, "Ubiquitous Nature of Event-Driven Approaches: A Retrospective View (Position Paper), *Proc. Dagstuhl Seminar 07191*, 2007.

[10] S. Chakravarthy, et al., "HiPAC: A research project in active, time-constrained database management", *Technical Report XAIT-89-02*, *Xerox Advanced Information Technology*, August 1989.

[11] S. Chakravarthy and D. Mishra, "Snoop: An Expressive Event Specification Language for Active databases", *Data and Knowledge Engineering(DKE)*, Vol. 14, pp. 1-26, 1994.

[12] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim, "Composite Events for Active Databases: Semantics, Contexts and Detection", *Proc. VLDB*, 1994.

[13] S. Chakravarthy, E. Anwar, L. Maugis, and D. Mishra, "Design of Sentinel: An Object-Oriented DBMS with Event-Based Rules", *Information*

and Software Technology, vol.36, no.9, pp.559-568, 1994.
[14] S. Chandrasekaran and M. J. Franklin, "Streaming Queries over Streaming Data", *Proc. VLDB* 2002.
[15] J. Chen, D. DeWitt, F. Tian, and Y. Wang, "NiagaraCQ: A Scalable Continuous Query System for Internet Databases", *Proc. SIGMOD* 2000.
[16] A. Demers, J. Gehrke, and B. Panda, "Cayuga: A General Purpose Event Monitoring System", *Proc. CIDR*, 2007.
[17] L. Elkhalifa, R. Adaikkalavan, and S. Chakravarthy: "InfoFilter: A System for Expressive Pattern Specification and Detection Over Text Streams", *Proc. SAC*, 2005.
[18] P. Fahy and S. Clarke, "CASS – a middleware for mobile context-aware applications", *Proc. Workshop of Context Awareness, MobiSys* 2004.
[19] C.L. Forgy, "Rete: A fast algorithm for the many pattern/many object pattern match problem", *Artificial Intelligence*, vol.19, no.1, pp.17-37, September 1982.
[20] V. Garg, R. Adaikkalavan, and S. Chakravarthy, "Extensions to Stream Processing Architecture for Supporting Event Processing", *Proc. DEXA*, 2006.
[21] S. Gatziu and K. R. Dittrich, " SAMOS: an Active Object-Oriented Database System", *IEEE Quarterly Bulletin on Data Engineering*, vol. 15, no. 1-4, December 1992.
[22] S. Gatziu and K. R. Dittrich, "Events in an Active Object-Oriented Database System", *Proc. Rules in Database Systems* (*RIDS*), 1993.
[23] N. Gehani, H. V. Jagadish, and O. Shmueli, "COMPOSE: A System for Composite Event Specification and Detection", *Advanced Database Systems*, *Lecture Notes in Computer Science*, vol. 759, pp.3-15, 1993.
[24] N. Gehani and H. V. Jagadish, "Ode as an Active Database: Constraints and Triggers", *Proc. VLDB*, 1991.
[25] L. Golab and M. Tamer Ozsu, "Data Stream Management Issues – A Survey", *SIGMOD Record* 2003.
[26] R. E. Gruber, B. Krishnamurthy, and E. Panagos, "The Architecture of the READY Event Notification Service", *Proc. ICDCS Workshop*, 1999.
[27] E. N. Hanson and T. Johnson, "Selection Predicate Indexing for Active Databases using Interval Skip Lists", *Information Systems*, vol. 21, no. 3, pp. 269–298, 1996.
[28] E. N. Hanson, M. Chaabouni, C. Kim, and Y. Wang, "A predicate matching algorithm for database rule systems", *Proc. SIGMOD* 1990.
[29] E. N. Hanson, C. Carnes, L. Huang, M.Konyala and L.Noronha, "Scalable Trigger Processing", *Proc. ICDE* 1999.
[30] E. N. Hanson, "The Design and Implementation of the Ariel Active Database Rule System", *IEEE Transactions on Knowledge and Data Engineering(TKDE)*, vol. 8, no. 1, February 1996.
[31] J. M. Hellerstein, W. Hong, S. Madden, and K. Stanek, "Beyond average: Towards Sophisticated Sensing with Queries", *Proc. IPSN*, 2003.
[32] A. Hinze and S. Bittner, "Efficient Distribution-Based Event Filtering", *Proc. DEBS workshop*, 2002.
[33] A. Hinze, "Efficient Filtering of Composite Events", *Proc. BNCD*, 2003.
[34] H. Hu, J. Xu and D. Lee, "A Generic Framework for Monitoring Continuous Spatial Queries over Moving Objects", *Proc. SIGMOD* 2005.
[35] Q. Jiang, R. Adaikkalavan, and S. Chakravarthy, "Estreams: Towards an Integrated Model for Event and Stream Processing", *Technical Report CSE-2004-3*, *University of Texas at Arlington*, July 2004.
[36] D. V. Kalashnikov, S. Prabhakar, W. G. Aref, and S. E. Hambrusch, "Efficient evaluation of continuous range queries on moving objects", *Proc. DEXA*, 2002.
[37] J. Kang, J. F. Naughton, and S. D. Viglas, "Evaluating Window Joins over Unbounded Streams", *Proc. ICDE*, 2003.
[38] Korea stock exchange. http://www.kse.or.kr.
[39] 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 CS-TR-2004-202*, August 2004.
[40] J. Lee, Y. Lee, S. Kang, S. Lee, H. Jin, B. Kim, and J. Song, "BMQ-Index: Shared and Incremental Processing of Border Monitoring Queries over Data Streams", *Proc.* MDM 2006.
[41] G. Li and H. Jacobsen, "Composite Subscriptions in Content-Based Publish/Subscribe Systems", *Proc. Middleware*, 2005.
[42] J. Liu, M. Chu, J. Liu, J. Reich, and F. Zhao, "State-Centric Programming for Sensor-Actuator Network Systems." *IEEE Pervasive Computing*, pp.50-62, October, 2003.
[43] C. Ma and J. Bacon, "COBEA: A CORBA-Based Event Architecture", *Proc. COOTS*, 1998.
[44] S. R. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman, "Continuously Adaptive Continuous Queries over Streams", *Proc. SIGMOD* 2002.
[45] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "The Design of an Acquisitional Query Processor for Sensor Networks", *Proc. SIGMOD* 2003.
[46] M. F.Mokbel, X. Xiong and W. G. Aref, "SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Database", *Proc. SIGMOD* 2004.
[47] M. F.Mokbel and W.G.Aref, "Generic and Progressive Processing of Mobile Queries over Mobile Data", *Proc. MDM* 2005.
[48] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma, "Query Processing, Resource Management, and Approximation in a Data Stream Management System ", *Proc. CIDR* 2003.
[49] Network-based Generator of Moving Objects, http://www.fh-oow.de/institute/ iapg/personen/brinkhoffgenerator/
[50] N. W. Paton and O. Diaz, "Active Database Systems", *ACM Computing Surveys*, vol. 31, no.1, pp. 63-103, March 1999.
[51] P. R. Pietzuch and J. Bacon, "Hermes: A Distributed Event-Based Middleware Architecture", *Proc. DEBS workshop*, 2002.
[52] P. R. Pietzuch, B. Shand, and J. Bacon, "A Framework for Event Composition in Distributed Systems", *Proc. Middleware*, 2003.
[53] S. Prahakar, Y. Xia, D. V.Kalashnikov, W. G.Aref and S. E. Hambrusche. "Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects", *IEEE Trans. Computers*, vol.51, no.10, pp. 1124-1140, 2002.
[54] M. A. Sharaf, J. Beaver, A. Labrinidis, and P. K. Chrysanthis, "TiNA: A Scheme for Temporal Coherency-Aware in-Network Aggregation", *Proc. MobiDE*, 2003.
[55] M. Srivastava, "Wireless Sensor and Actuator Networks", Tutorial, *Proc. MOBICOM* 2005.
[56] K. Terfloth, G. Wittenburg, and J. Schiller, "FACTS – A Rule-based Middleware Architecture for Wireless Sensor Network", *Proc. COMSWARE*, 2006.
[57] University of Washington. Live from Earth and Mars. http://www-k12.atmos.washington.edu/k12/ grayskies/ nw_weather.html
[58] S. Urban, I. Biswas, and S. W. Dietrich, "Filtering Features for a Composite Event Definition Language", *Proc. SAINT*, 2006.
[59] J. Widom, "The Starburst Active Database Rule System", *IEEE Transactions on Knowledge and Data Engineering(TKDE)*, vol. 8, no. 4, August 1996.
[60] G. Wittenburg, K. Terfloth, F. L. Villafuerte, T. Naumowicz, H. Ritter, and J. Schiller, "Fence Monitoring - Experimental Evaluation of a Use Case for Wireless Sensor Networks", *Proc. EWSN* 2007.
[61] K. L. Wu, S. Chen and P. S. Yu, "Indexing Continual Range Queries for Location-Aware Mobile Services", *Proc. EEE* 2004.
[62] K. L. Wu, S. Chen and P. S. Yu, "On Incremental Processing of Continual Range Queries for Location-Aware Services and Applications", *Proc. MobiQuitous* 2005.
[63] K. L. Wu and P. S. Yu, "Interval Query Indexing for Efficient Stream Processing", *Proc. CIKM*, 2004.
[64] E. Wu, Y. Diao, and S. Rizvi, "High-Performance Complex Event Processing over Streams", *Proc. SIGMOD*, 2006.
[65] Y. Yao and J. Gehrke, "Query processing for sensor networks," *Proc. CIDR*, 2003.
[66] S. Yoon and C. Shahabi, "The Clustered AGgregation (CAG) Techniques Leveraging Spatial and Temporal Correlations in Wireless Sensor Networks", *ACM Transactions on Sensor Networks*, vol. 3, no. 1, March 2007.