Optimization of Multiple Continuous Queries over RFID Streaming Data

Haipeng Zhang, Wooseok Ryu, Md. Kafil Uddin, Bonghee Hong
Department of Computer Engineering
Pusan National University
Pusan, Republic of Korea
E-mail: {jsjzhp, wsryu, mdkafil, bhhong}@pusan.ac.kr

Abstract—**RFID technology enables a new era of business optimization. With the development of RFID technology, more and more RFID applications have been developed. In RFID system, RFID middleware collects, filters, and integrates large volume of streaming data gathered continuously by heterogeneous readers to process queries from applications. These queries are called continuous queries as they are executed continuously to extract useful information from data streams. EPCglobal proposed an Event Cycle Specification (*ECSpec*) model, which is a de facto standard query interface for RFID middleware. When the middleware system processes many continuous queries, query optimization is quite important for their execution and enhancing the performance of the system. In this paper, we propose a multiple continuous query optimization method for RFID data streams, which is based on query (*ECSpec*) execution conditions and filter conditions analysis.**

Keywords-RFID middleware; Continuous queries; Query optimization.

I. INTRODUCTION

The Radio Frequency Identification (RFID) as a frontier technology is an automatic identification and data capture technology that uses RF waves to transfer data between a reader and a tag which attached on an object. RFID provides fast, reliable, and automatic identifying, locating, tracking and monitoring physical objects without line of sight. With such benefits, RFID is gradually being adopted and deployed in various applications, such as supply chain management systems, warehouses management, assets tracking, and ubiquitous computing applications, etc.

RFID system mainly consists of four components: tags, RFID transceivers or readers, RFID middleware, and RFID application [1, 2]. Tags store the unique ID (EPC code) and related data in their memory to uniquely identify the object. Readers are used for reading the information stored at RFID tags placed in their interrogator zone and sending the streaming data to middleware through wired or wireless interfaces. Middleware systems collect data from readers, process receiving data according to the requests of applications, and generate reports sending to applications. Applications are the software components which issue requests to middleware and provide business services to users, such as supply chain management or warehouses management.

In the RFID system, middleware should process the high volume of raw RFID streaming data to reply the requests of applications. These requests are called continuous queries [3, 4] because they are executed continuously to extract information from data streams. These queries are usually triggered by special events such as the arrival of new data items from data streams or system timer alarms, get useful information from data streams by filtering, compose new information joining multiple data streams, and send query results to users. When users register many continuous queries, middleware system processes these queries continuous over large volume of streaming data. It may result the response delay or even burden for middleware, so multiple query optimization is quite important for efficient execution and enhancing the performance of middleware system.

For collecting, filtering, and grouping RFID streaming data, EPCglobal proposed an Event Cycle Specification (*ECSpec**) model [5], which is a de facto standard query interface for RFID middleware. *ECSpec* can be treated as continuous query defined by users and continuously executed in middleware, which has execution and filter conditions. Once an *ECSpec* defined by user, it will be subject to a lifecycle state transition specified in the *ECSpec*. An *ECSpec* must specify start conditions and stop conditions which together define a time interval to extract information of interest from data streams and generate the results. Queries having the same operators may share a lot of intermediate results when they are executed at close instants, but may involve only disjoint data when executed at completely different instants. So, query execution timing as well as common query predicates is a key to deciding an efficient query execution plan. In this paper, we analysis the properties of *ECSpec* to identify the execution patterns and query conditions of it. Such patterns offer useful information for optimizing the execution of multiple continuous queries. By using this information, we form clusters of continuous queries such that queries in the same cluster are likely to share the intermediate result, extract common conditions from queries in each cluster, and decide the optimal query execution plan.

The remainder of this paper is organized as follows. Section 2 presents some related work on continuous queries optimization. Section 3 analyzes and explains ALE and RFID continuous query execution model (*ECSpec*). Section 4 presents the proposed query optimization technique. Finally, Section 6 concludes the paper with the future work.

* We use the term *ECSpec* and query interchangeably throughout the paper.

II. RELATED WORKS

There are many research efforts have been made on continuous queries processing and continuous queries optimization. OPenCQ[4] is a system integrating distributed heterogeneous information sources and supports continuous queries. Continuous queries in OpenCQ consist of three parts: query conditions, trigger condition, and terminal condition. When the trigger condition is satisfied, the query is executed continuously until the terminal condition is satisfied. However, sophisticated multiple query optimization is not addressed.

NiagaraCQ [6, 7] proposes a multiple query optimization method for its continuous queries. It can handle large-scale queries and supports incremental multiple queries optimization. Simple selection predicates are grouped by their expression signatures and evaluated in chains. However, continuous queries in NiagaraCQ are simple and do not use window operators to specify time intervals of interest as in the window join, which is not suitable for extracting useful information from RFID data streams.

TelegraphCQ [8] is another system designed to process a large number of continuous queries. Based on eddy [9], it realizes adaptive processing, dynamically reordering operators to cope with changes of arriving data properties and selectivity, and supporting multiple queries optimization by grouping and indexing individual predicates. CACQ evaluates queries aggressively; it picks up operators as soon as they become executable and evaluates them immediately. In cases where queries are associated with window-based time intervals and/or the execution time specifications, this query evaluation method may generate more redundant query results than are needed.

ARGUS [10] is a stream processing system implemented atop commercial DBMSs to support large-scale complex continuous queries over data streams. It supports incremental operator evaluation and incremental multiple query plan optimization as new queries arrive. It builds a whole query network for all the queries, which costs time to maintain a big query network.

There are some other works also related to data stream processing. In [11] it split and merge query conditions, and sends the common condition to the reader level to reduce the duplicated data. It doesn't consider sharing of the intermediated data and only supports the reader implemented with RP protocol [12]. STREAM [13] is another continuous queries architecture which focuses on developing execution engine, with emphasis on incremental evaluation methods and adaptive processing on scheduling and approximate answers. However it does not address queries optimization.

Traditional relational queries optimization schemes were originally proposed in [14, 15]. Basically, they concentrate on extracting common sub-expressions from among multiple queries to share intermediate query results. In our approach, we analyze the query execution patterns to form clusters of continuous queries in which common query conditions can be extract and get the efficient query execution plan.

III. APPLICATION LEVEL EVENTS

In RFID systems, readers read tags for a very large volume of streaming data; however, the raw data generated is of a low level and is not good to directly used for applications. These applications require answers to specific questions from the streaming data. So, there need a level of processing that reduces the volume of data that comes directly from readers into coarser "event" of interest to applications. This led EPCglobal to develop the Application Level Events (ALE) Specification which is the de facto standard interface for filtering, grouping, counting RFID streaming data and reporting to applications in various forms.

Through the ALE interface, applications may define and manage event cycle specifications (*ECSpecs*). This interaction may take place in a "pull" mode, where users provide the *ECSpec* and the ALE in turn initiates or waits for read events, processes the data, and returns the report. In this case, *ECSpec* can be considered as one time queries. On the other hand, it may also be done in a "push" mode, where the client registers a subscription to a defined *ECSpec*, and thereafter the ALE asynchronously sends reports to the applications when event cycles complete. In this case, *ECSpec* can be considered as a continuous query. Fig. 1 shows the operation description of ALE. After an *ECSpec* defined by users, it can be subscribed and continuous executed. When the ALE server receives users' subscriptions, it first analyzes the *ECSpec*, and then starts to receive EPC data from data sources. This data is collected in every reader cycle, filtered by filter conditions and organized in groups in every event cycle. Finally, reports will be generated and sent to users.

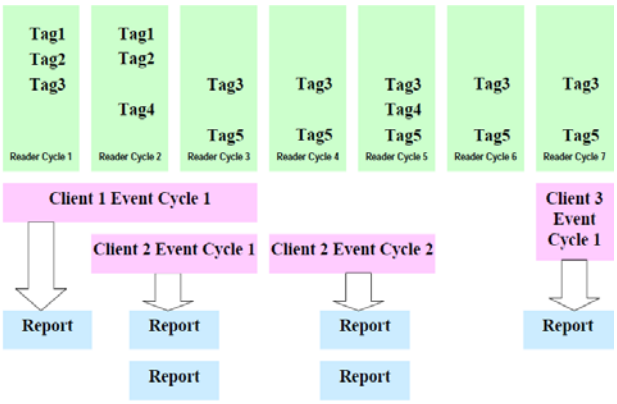

Figure 1. The operation description of ALE

An *ECSpec* describes an event cycle and one or more reports that are to be generated from it. It contains three main parts which are a list of logical readers whose data are to be included in the event cycle, a specification of how the boundaries of event cycles are to be determined, and a list of specifications each of which describes a report to be generated from this event cycle (refer to Fig. 2). An *ECBoundarySpec* specifies how the beginning and end of event cycles are to be determined; it controls that how to execute the *ECSpec*. In order to get the results from infinite RFID data streams, *ECBoundarySpec* defines time windows in which data is processed and results are generated at the end of each window. Time windows can be determined by start and stop conditions.

The condition of logical readers represent that the data stream generated from which reader should be processed. Each *ECReportSpec* contains an *ECFilterSpec* which represents what tags are to be included in the report.

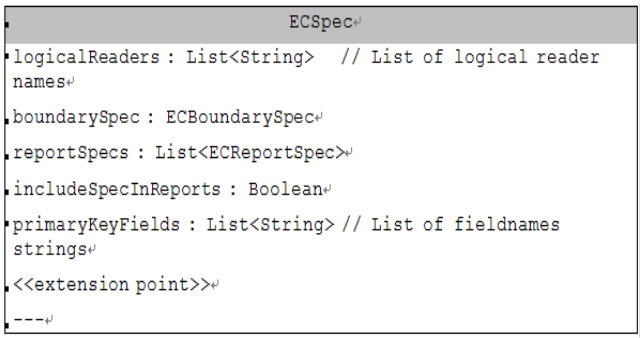

Figure 2. ECSpec

ALE can support a lot of user clients and connect many heterogeneous readers, so it may receive large volume of streaming data generated by readers and many *ECSpecs* defined by users. These *ECSpecs* continuously are executed in ALE that may cause the response delay or reduce the performance of ALE system. So queries optimization is a very important method for continuous queries processing to alleviate the system burden and get results in time. As we know, if queries having the same operator may share intermediate result when they are executed at close instants. For example, in Fig. 1, the execution time of client 1 event cycle 1 is close to client 2 event cycle 1, and the window range referred by client 1 event cycle overlaps the range of client 2 event cycle 1. However, client 3 event cycle 3 is separated from other's. And if having some common operators, the intermediate results generated within client 1 event cycle 1 contains the same results generated within client 2 event cycle 2 as their execution time intervals overlap with each other. Therefore, the results of client 1 can be shared with client 2. The more intermediate results can be shared, the more system load can be saved. Thus, intermediate results can be shared or not definitely affect the effectiveness of multiple continuous queries optimization. Our proposed method of continuous queries optimization is represented in the next section.

IV. CONTINOUS QUERIES OPTIMIZATION

As described above, continuous queries having some common operators can share their intermediate results in case of their have close execution time. In this section, we analyze execution patterns of *ECSpec* and define similarity to form clusters whose members have close execution times and large overlaps of time intervals; and then extract common filter conditions for each cluster and decide the optimal query execution plan.

A. Query Execution Pattern Analysis

In order to get the queries that have close execution times, we need analyze query execution patterns. As mentioned in Section 3, *ECBoundarySpec* (Fig. 3) specifies the execution conditions of the *ECSpec*. The *startTrigger* and *stopTrigger* define triggers that may start a new event cycle or stop an event cycle, respectively. The *repeatPeriod* parameter specifies an interval of time for starting a new event cycle, relative to the start of the previous event cycle. The *duration* parameter specifies an interval of time for stopping an event cycle, relative to the start of the event cycle. The last two parameters specify stopping an event cycle. These two parameters are not considered in this paper since they are rarely used. According to ALE specification, a *ECTrigger* takes one main following form: *urn:epcglobal:ale:trigger:rtc:period.offset*. A trigger of this form means that it is delivered each time the number of milliseconds past midnight modulo *period* equals *offset*. The *period* defines a time period, in milliseconds between consecutive triggers occurring within one day within the range $1 \leqslant period \leqslant 86400000$. The *offset* defines a time interval in milliseconds between midnight and the first trigger and the first trigger delivered after midnight, and it must less than the specified *period*. For example, the following trigger denotes a trigger that occurs every hour on the hour: *urn:epcglobal:ale:trigger:rtc:3600000.0*, that means it continuously occurs at 0:00, 01:00, 02:00, etc.

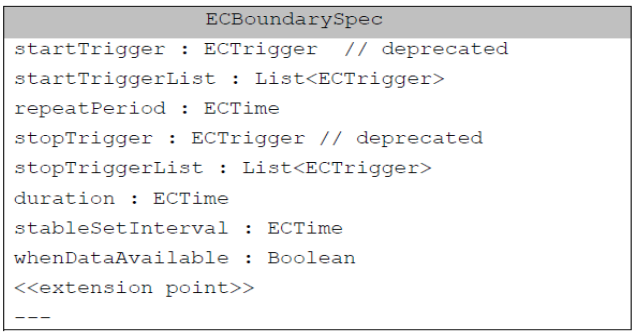

Figure 3. ECBoundarySpec

An event cycle begins when the first start condition (repeat period or one of the start triggers) occurs. If no start triggers are specified, the first event cycle begins immediately after client's subscription of the *ECSpec*. The start conditions have no affect on an event cycle which is in progress. The event cycle terminates only when one of the stopping conditions (duration or one of the stop triggers) specified above becomes true. Thus, *ECBoundarySpec* must contain at least one stop condition. An execution example of event cycle is shown in Fig. 4. *RepeatPeriod* and *duration* are specified in the *ECBoundarySpec*, the first event cycle begins immediately after client's subscription arrived and terminates when the *duration* expires. Then, another event cycle begins when the *repeatPeriod* has expired. The *ECSpec* is continuously executed in this way until client unsubscribes it.

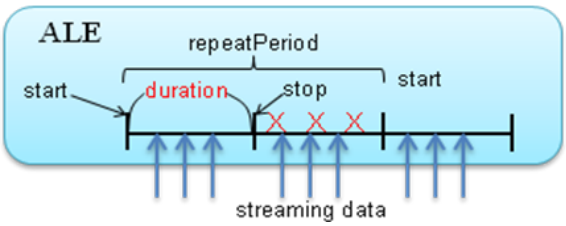

Figure 4. An execution example

B. Query Model Analysis

ALE specifies *ECSpec* as a standard query for RFID middleware which contains several filter predicates related to tags and readers. *LogicalReaders* is one of them which contain at least one logical reader name. Each logical reader contains one or more physical reader names and defines filter condition for readers. Another important predicates is *ECFilterSpec* which specifes what tags are to be included in the final report. It contains a set of filter list members. Each filter list member consists of three parameters: *includeExclude*, *fieldSpec*, and *patList* (refer to Fig. 5).

```
ECFilterListMember
includeExclude : ECIncludeExclude // (INCLUDE or EXCLUDE)
fieldspec : ECFieldSpec
patList : List<String>  // one or more patterns
<<extension point>>
---
```

Figure 5. ECFilterListMember

The *fieldspec* specifies which field of the tag is considered to evaluate this filter, and the format for patterns in the *patList*. The field usually specifies *epc* field that contains EPC codes. The value of the *includeExclude* is INCLUDE or EXCLUDE. If the value is INCLUDE, a tag is considered to pass the filter if the value of the field matches at least one pattern specified in the *patList*. If EXCLUDE, a tag is considered to pass the filter if the value of the field doesn't matches all the patterns specified in the *patList*.

The *patList* specifies the patterns to compare with the specified tag field. There are four types of valid patterns: *fixValue*, *, *[lo-hi]*, and *&mask=value*.If a pattern is a single value (*fixValue*), the pattern matches a value equal to the pattern. If a pattern is the *, the pattern matches any value. If a pattern is in the form *[lo-hi]*, the pattern matches any value between *lo* and *hi*, inclusive. If a pattern is in the form *&mask=value*, the pattern matches any value that is equal *value* after being bitwise and-ed with *mask*. The two patterns * and *[lo-hi]* specify a range of data matching them. If one pattern covers fully or partially some space of another pattern, the overlapped part can be extracted and shared by these patterns. For example, p_1*=[3-10]* and p_2*=[6-15]* are overlapped, and *[6-10]* is the common pattern of them.

C. Cluster Queries

According to the analysis above, we know that the start conditions and stop conditions specify execution time intervals of the query. So, once a query defined, we can get continuous time intervals from it. Thus, according to these time intervals, we can know which queries may share their intermediate results if they have close executing time instances and overlapped time intervals. In the following part, we introduce how to detect queries that have large overlapped time intervals.

After some *ECSpecs* defined, we can get a set of execution time intervals of each *ECSpec*. Since we do not know when queries will be unsubscribed and in order to easily get the execution time intervals, we suppose the queries cannot be unsubscribed before the midnight since they begin executing. In this case, we can extract the execution time intervals of each query within a certain time span *T* which from the start time of query to the midnight. After we get the set of execution time intervals, we can detect which queries have overlaps of execution time intervals. To achieve this, we cluster queries so that queries in the same cluster have large overlaps of interval which maybe share their intermediate results. In order to cluster the queries, the similarity of queries is needed to define. We define the *unit-length interval* on one day's time and label them sequentially. The length of *unit-length interval* is not fixed and changeable. For example, we can set it as 10 minutes, 30 minutes, 1hours, etc. Now, for each query Q_i, we can get a set of T_i whose elements are sequence numbers of *unit-length intervals*. T_i can be computed as follows:

$$T_i = \bigcup_{e \in ET_i} \{ t \in e \wedge t \in UI \} \tag{1}$$

Where ET_i is the set of execution time interval of query Q_i, *UI* is the labeling number set of *unit-length intervals*.

Then the *similarity* (Q_i, Q_j) can be computed as follows:

$$similarity(Q_i, Q_j) = \frac{|T_i \cap T_j|}{|T_i \cup T_j|} \tag{2}$$

The similarity of queries measures how much overlap exists between two queries. If the similarity equals to 0, it means these two queries have no overlaps of time intervals. If the similarity equals to 1, it means these two queries have the same execution time intervals. In order to form clusters, we need compute similarity of all queries.

Using the similarity of all queries, we can form cluster that have the high similarities. To generate the clusters, we use a hierarchical clustering algorithm, where queries in the same cluster have similarity more than or equal to a predefined threshold θ ($0 \le \theta \le 1$). The size of the generated clusters is decided by the threshold which is also the criterion for deciding how many overlaps of execution time intervals the queries have. If θ equals to 0, it means all the queries form only one cluster. If θ equals to 1, it means only queries having same time intervals form a cluster.

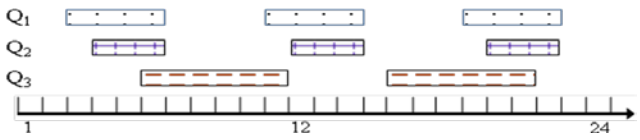

Figure 6. Queries execution time intervals

For an example, there are three queries, and their execution time intervals are shown in Fig. 6. We set the unit-length interval to 1 hour and label them from 1 to 24. So, we can compute time interval set of each query as follow:

T_1 *= {3, 4, 5, 6, 11, 12, 13, 14, 19, 20, 21, 22}*

T_2 *= {4, 5, 6, 12, 13, 14, 20, 21, 22}*

$T_3 = \{6, 7, 8, 9, 10, 11, 16, 17, 18, 19, 20, 21\}$

Using Ti, we can compute the similarities of each query: $similarity(Q_1, Q_2) = 0.75$, $similarity(Q_1, Q_3) = 0.22$, $similarity(Q_2, Q_3) = 0.17$. If the threshold θ set to 0.6, we can get two clusters: Q_1 and Q_2 belong to a cluster, Q_3 belongs to another cluster.

After clusters formed, they need to update since during system running queries insertion and deletion may occur. In this case, we need to reconstruct clusters to maintain the best similarity. However, this process is too expensive. To provide high performance, we reconstruct clusters only when the number of queries changes more than a predefined constant value. For insertion, a new coming query is added to the cluster having the query with highest similarity. For deletion, just remove query from the cluster.

D. Query Optimization

As previous description, queries in the same cluster have high similarities. So they can share some intermediate results generated by common query filtering conditions. Therefore, we need find out common query conditions and generate optimal query plan for each cluster. To get the common conditions of queries in the cluster, we refer to the algorithm proposed in [11] to get common query conditions. We use its algorithms to split and merge query filter conditions to find the common conditions for queries in each cluster. Using these common conditions, optimal query plan can be derived. We choose the optimal query plan that can share common filter conditions as more as possible. For example, there are there queries Q_1, Q_2, and Q_3 in the same cluster shown in table 1.

TABLE I. EXAMPLE OF QUERIES

Query	Reader Specification	Filter Condition
Q_1	R_2	${}^1PatList_1 = <[10\text{-}15]>$
Q_2	R_1	${}^2PatList_1 = <[10\text{-}13]>$
Q_3	R_1	${}^3PatList_1 = <[6\text{-}14]>$

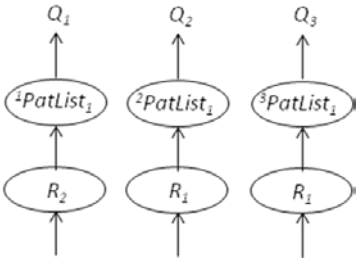

Figure 7. Query execution plans

Here we suppose all the patterns are EXCLUDE. Fig. 7 shows one possible execution plan for three queries. According to the filter conditions, we know they have overlapped conditions. So we split the patterns and find the overlapped pattern. The patterns can be split as follows:

${}^1PatList_2 = <[10\text{-}13]>, {}^1PatList_3 = <15>$

${}^2PatList_1 = <[10\text{-}13]>$

${}^3PatList_2 = <[6\text{-}9]>, {}^3PatList_3 = <[10\text{-}13]>$

${}^3PatList_4 = <14>$

From split patterns, we know filter condition <[10-13]> is the common condition of the three queries. So, we can change the query plan by using new patterns. Fig. 8 shows the new query execution plan for three queries. Comparing these two execution plans, the second one is better than the previous query execution plans shown in Fig. 7. Since there are two query conditions are shared in the optimal execution plan. In this case, it can reduce duplicated data and save processing time. So, our query optimization method can share query operators, reduce redundant result, save response time, and enhance performance of ALE middleware.

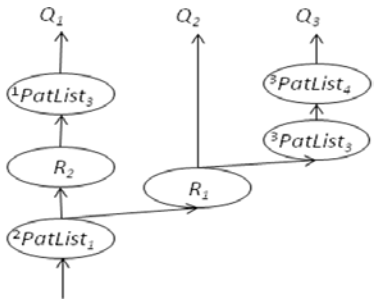

Figure 8. An optimal execution plan

V. CONCLUSIONS

RFID middleware systems process high volume of raw RFID streaming data to reply the requests of applications. To process RFID data, EPCglobal has specified a de facto standard query interface for RFID middleware. In this paper, we proposed an optimization approach to execute continuous queries for RFID middleware. In our approach, it first computes similarities of queries based on queries execution time intervals; then forms query clusters having high similarities; at last for each cluster, finds the common filter conditions and generates optimal query plans for each cluster. The optimal query plan has the most common query conditions to share intermediate results. So our approach can well used for ALE to process RFID streaming data.

ACKNOWLEDGMENT

This work was supported by the Grant of the Korean Ministry of Education, Science and Technology (The Regional Core Research Program/Institute of Logistics Information Technology).

REFERENCES

[1] K. Finkenzeller. RFID handbook. Wiley Hoboken, NJ, 2003.

[2] B. Glover and H. Bhatt. RFID Essentials. O'Reilly Media, Inc. 2006.

[3] D. Terry, D. Goldberg, and D. Nichols. "Continuous Queries over Append-Only Databases," Proc, ACM SIGMOD, 1992, pp. 321-330.

[4] L. Liu, C. Pu, and W. Tang. "Continual Queries for Internet Scale Event-Driven Information Delivery," IEEE TKDE, vol.11, no.4, 1999, pp. 610-628.

[5] The Application Level Events (ALE) Standard Version 1.1. http://www.epcglobalinc.org/standards/ale/ale_1_1_1-standard-core-20090313.pdf.

[6] J. Chen, D.J. DeWitt, F. Tian, and Y. Wang. “NiagaraCQ: A Scalable Continuous Query System for Internet Databases,” Proc. ACM SIGMOD, 2000, pp. 379–390.

[7] J. Chen, D.J. DeWitt, and J.F. Naughton. “Design and Evaluation of Alternative Selection Placement Strategies in Optimizing Continuous Queries,” Proc. ICDE, 2002, pp. 345–356.

[8] S. Madden, M. Shah, J.M. Hellerstein, and V. Raman. “Continuously Adaptive Continuous Queries over Streams,” Proc. ACM SIGMOD, 2002, pp. 49–60.

[9] R. Avnur, and J.M. Hellerstein. “Eddies: Continuously Adaptive Query Processing,” Proc. ACM SIGMOD, 2000, pp. 261–272.

[10] C. Jin, J. Carbonell. “ARGUS: Efficient Scalable Continuous Query Optimization for Large-Volume Data Streams,” IEEE IDEAS, 2006, pp. 256-262.

[11] M. Ashad, R. Wooseok, and H. BongHee.S. “Reader Level Filtering for Query Processing in an RFID Middleware,” The Institute of Electronics Engineers of Korea, Vol. 45 C1 No. 3, pp. 113-121.

[12] The Reader Protocol Standard Version 1.1. http://www.epcglobalinc.org/standards/rp/rp_1_1-standard-20060621.pdf.

[13] R. Motwani, J. Widom, and A. Arasu, etc. “Query Processing, Resource Management, and Approximation in a Data Stream Management System,” Proc. CIDR, 2003, pp. 245-256.

[14] T.K. Sellis. “Multiple-Query Optimization,” ACM TODS, vol.13, no.1, 1988, pp. 23–52.

[15] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. “Efficient and Extensible Algorithms for Multi Query Optimization,” Proc. ACM SIGMOD, 2000, pp. 249–260.