A Combination Approach for Distributed Information Flow Processing in Multi-purpose IoT

Sunyanan Choochotkaew[1,a)] Hirozumi Yamaguchi[1,b)] Teruo Higashino[1,c)]
Megumi Shibuya[2,d)] Teruyuki Hasegawa[2,e)]

Abstract: A study to process bursting information flowing around is continuously developed for long time and is still ongoing under Information Flow Processing (IFP) terminology. There are various types of processing goals such as for tracking some circumstance, extracting any extra knowledge or detecting some critical events. Until now, there are two main ways to implement the processing engine: Data Stream (DSMS) and Complex Event Processing (CEP). Each approach is designed for different tasks. The former is superior for processing relational relation among data in the stream while the latter is superior for detecting specific events. To support multi-purpose IoT, we also define a new language covering the most expressive language in both approaches. In the second place,we propose a new IFP engine that exploits advantage points from both approaches: relational operations in structural database and matching operations in powerful rule engine in distributed-deployment architecture. In the final part, we preliminary test our implementation design on Java streaming in a number of various-type data. We give examples of superior and inferior subscriptions for the rule-based approach and the relational-stream-based approach and compare our combination methods to relational-stream-based approach against each example subscription.

1. Introduction

Nowadays, people become aware of an extended concept from a machine-to-machine communication to an Internet of Things (IoT) as an important role for improving human life. Without human intervention, physical things observe themselves as well as their surroundings, process those information and then might activate some actions to serve human needs without the boundary of just one network.

One of the most challenging issues to realize IoT concept is the way to deal with the information flow with large volume, fast velocity, and high variety. A pile of research is proposed and arranged under the topic called information flow processing (IFP) [5]. It is mainly classified into two categories: Data Stream Management System (DSMS) and Complex Event Processing (CEP). According to [5], there are many ways to classify but the most significant one is the data-model aspect. From the viewpoint of this aspect, the former manipulates generic-data streams while the latter deals with event notifications. To the best of our knowledge, the most expressive language for DSMS is CQL [1] while the common language for CEP is TESLA[3].

Both DSMS and CEP are developed almost independently. Each of them has its own proper use; however, it could not support the other efficiently. Although there are some work that add a simple event detection feature to the DSMS system, they, still, are not available for complex event detection as good as the absolute CEP system.

Complex event processing (CEP) considers data in the flow independently not as a stream to detect the interesting events in parallel. On the other hand, it has to keep temporary copies of partial detected events in nonstructural manner. Consequently, the relational processing including aggregation, parametrization (join) consumes extra cost comparing to the stream-based approach. Also, some operations such as outer-join, grouping are not defined in TESLA. Furthermore, in the automaton-based engine, the same event is likely to remain in multiple automata for the same subscription [4]; meanwhile, rule-based engine mostly keeps the partial detected event in the same working memory. As a result, those partial detected events have to activate every time a new event comes even though it hardly has a chance to be matched [2]. Accordingly, CEP is proper for detecting, but not for processing relational relation among events.

On the contrary, stream-based approach (DSMS) can support all relational operations efficiently because of the structural database [1]. However, the detection must be performed in order as well. In addition, it requires windowing process before starts processing. All data have to be organized structurally even though it will never be needed if it was tested with some condition first. Thus, DSMS is designed for massively processing relation relations among data which flowing as streams not specific to some detection.

[1] Graduation School of Information Science and Technology Osaka University, Osaka, Japan
[2] KDDI R&D Laboratories, Saitama, Japan
[†1] Presently with DICOMO2016
[a)] sunya-ch@ist.osaka-u.ac.jp
[b)] h-yamagu@ist.osaka-u.ac.jp
[c)] higashino@ist.osaka-u.ac.jp
[d)] shibuya@kddilabs.jp
[e)] teru@kddilabs.jp

In the multi-purpose IoT system, a various kind of processing is required. For instance, in smart home system, a stream process is needed for energy saving while an complex event like fall detection is also needed for monitoring elderly people. To achieve all purposes with just one IFP engine, we design a new framework concerning advantage points of both approaches: parallel detection and relational operation. We consider information flows as streams of events. All flowing events will be tested with defined conditions before memorized into the relational database for further processing relational operations.

To support our general purpose IFP engine, we analyze the overlapping between TESLA and CQL and, then, define a new event-definition language based on TESLA outline structure. Besides, windowing operation as well as relational operations including outer join and grouping are added.

Generally, IFP can be deployed in both centralized and distributed manner. For the latter one, it can be further classified into two groups: clustered and networked [5]. For better scalability, higher tolerance and more available mobility, we design our system in a distributed manners, specifically networked. Basically, we apply the publish-subscribe protocol, the rule-matching mechanism with simple relational database.

Our proposed engine is designed on the application layer with the knowledge of one-hop neighbors. In the deployment we implement our engine in Java over an ad-hoc network running B.A.T.M.A.N. routing protocol in Layer 2.

2. Related Works

An Information Flow Processing (IFP) terminology covers a wide area of processing continuous and timely information from multiple peripheral nodes responding to some points of the system which is hard to be accomplished efficiently by the traditional database management systems (e.g. TinyDB [10]). A long history of evolution in this field is primarily summarized in [5]. Broadly, there are two kinds of solutions: Data Stream Management Systems and Complex Event Processing. The significantly differentiating characteristic is the viewpoint of information flow. The former considers as multiple streams of data while the latter considers as multiple events.

2.1 Data Stream Management Systems: DSMSs

Instead of working on infrequently-updated data, Data Stream Management Systems (DSMSs) are specially designed for continuously-updated data. However, most of query representations in DSMSs are still based on SQL which used in traditional DBMSs including Continuous Query Language (**CQL**) which is the most expressive language for data stream applications according to the equivalence proofs shown in [1]. CQL is firstly supported by Standford DSMS, named *STREAM*, and now applied in the Oracle CEP runtime component.

To handle streams in a relation manner, CQL provides three groups of operations: Stream-to-Relation, Relation-to-Relation, and Relation-to-Stream. The first one converts a stream to relation by windowing with the range of time or the number of rows as well as partitioning by some attributes (i.e. grouping). Then, most of basic operations in traditional database are supported in the second operation groups. Finally, the operated results can be converted back to stream by the last operation group by creating *insertstream* (Istream), *deletionstream* (Dstream), or *relationstream* (Rstream).

Many DSMSs have focused on efficiency of processing multiple data streams with complicated operations to extract some knowledge from those streams in real-time. On the other hand, most of them cannot support complex event detections which are dependent on historical records. Besides, simple detections in the DSMSs have not done efficiently. For instance, there is no reuse of similar detections in the system and mostly done in a centralized manner.

2.2 Complex Event Processing: CEP

Whereas DSMSs flows information as data tuples, Complex Event Processing systems (CEP) flows that as event notifications. According to [5], CEP could be concluded that it is originated from publish-subscribe domain and then extends functionality by improving expressive power of subscriptions. Similar to the general system, there are two main ways of deployments: centralized and distributed. In earlier studies, distributed solutions usually limit powerfulness of complicated processing compared to centralized solutions. Conversely, the rapidly growing scale of sensing nodes also make centralized system weighty. As the best of our surveys, a formally-defined-event-specification language called **TESLA** [3] is mostly referenced in many proposals to represent powerfulness of processing complicated composite events.

- **Centralized deployment** One of the most important advantages of centralization in CEP is that, it allows system to compute any composite event easily no matter how complicated they are. Because all information are clearly available at the central node, the relations between them are also easy to be discovered at one point. As the best to our survey, the following solution is the most outstanding one.
  - *T-Rex:* Among a pile of researches, *T-Rex* is a centralized solution which combines expressiveness and efficiency [4]. It is likely-firstly adopts TESLA while provides efficient automaton-based-algorithm for event detection. Its evaluation-results show that throughputs significantly depends on complexity of the events.
- **Distributed deployment** While some researchers aim at powerfulness of complicated event processing, there are some researches more concern about the scalability, single point of failure, as well as over workload issue due to centralization. We would like to bring up some significantly contributing works as follows:
  - *PADRES:* As the best of our observation, a PADRES system is proposed as a distributed publish/subscribe

systems dealing with composite subscriptions at prior time. It consists of a brokers distributively deployed with static binary-tree topology [7]. Subscriptions are mapped to rules and publications are mapped to facts feeding to rule-based engine. Composite subscriptions are trivially decomposed from knowledge of advertisements of publication and tree-topology. Nevertheless, aggregate operations are not mentioned. Besides, the complex event such as an event of event or an event-dependent event are not supported.

– *RACED:* With regard to those indistinct issues, there are many studies further extend PADRES concepts in various aspects. RACED widens expressiveness by adopting TESLA for event definition and make available of complex event detections in distributed manner [2]. Likewise, it still straightforwardly decompose the composite subscriptions based on publication advertisements and tree-based topology. Nonetheless, it isn't limited to only binary topology as in PADRES. Besides, the authors also proposed a master-slave subscription protocol to reduce the number of unimportant packets in the network by letting children with slave subscriptions wait for child who holding master subscription to submit first.
– *Adaptive Content-Based Routing in General Overlay Topologies:* In the meanwhile, there are some proposals specially focuses on the limitation of tree topology of PADRES. An adaptive content-based routing protocol in general overlay topologies is proposed in [8] to handle cyclic and dynamic topology. In addition, the authors also suggest a cost model to determine whether decompose the composite subscriptions or not based on in-node and network conditions along with cardinality of subscriptions themselves.
– *DistCED:* Prior to PADRES, there is a useful general framework proposed for event composition in distributed system with defined language named CE [11]. Nodes in the systems is called CE detector. This framework sets a goal at large-scale application with existing of event patterns (i.e. event of event). The authors also suggest the way to define distribution policies as well as detection policies by addressing several dimensions. Still, it was only a framework without real implementation. Also, the defined language, CE, is less expressive than the current TESLA one.

2.3 Co-existing concept

For multi-purpose IoT system, both kinds of processing (i.e. DSMSs and CEP) are generally required for on the same information flows. There are many works realize the scenario where both kind of information flows are co-existing. RushNet prioritizes event-notification flows over general massive flows [9]. *StreamBase*, a commercial software run by TIBCO company provides event detections with different modules of relational query on the data stream [12].

However, to the best of my knowledge, the existing methods with co-existing concept never exploit the powerfulness of rule engine for detecting the simple conditions declaring by the event itself. All query is done in the relational database. While the approaches that uses such as parallel comparison engine like Padres, Raced, and T-Rex, are restrained from relational operations.

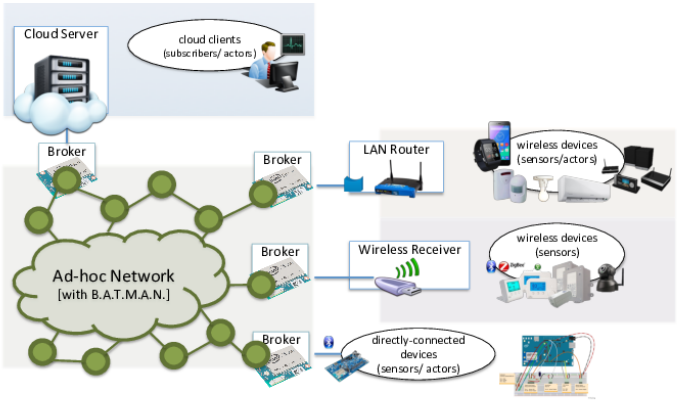

Fig. 1 Overview of Designed IoT Communication Network

We take an advantage of rule engine for primarily detecting the independent condition of each event to allow only worth-to-be-processed data flow into the relational database. With these idea, we remains benefit of the both approaches for Information Flow Processing in the multi-purpose IoT system. Even more, we also define a new combination language for our engine covering both TESLA and CQL in an expressive way.

3. System Design

A Flow-driven Distributed Information Flow Processing System is designed for Multi-purpose IoT environment where flows of information in the networks can vary with the user specifications. Our design principles is also concerned about scalability, dynamicity and mobility. We limit the constraints of pre-knowledges and static installation. The system is composed of many powerful mobile brokers to process information flowing in the network.

We adopt a concept of content-based publish/subscribe architecture where subscriber is system client and publisher is sensing device. However, instead of trivially returning a matched publication to subscriber, we extent the action part to allow subscriber to specify what to do with the matched results and who is the final actor to get those processed results (i.e. the final actor is not limited to the subscriber).

All brokers hold the same information of subscription and play the same role of processing. As a consequence, the simple matching process could be done at the point of publishing no matter where it happens. To handle overload processing on any one node, we propose a flow-driven analysis model to decide which publication could be forwarded with just small changes of the information flows in the networks.

3.1 Network Architecture Design

An overview architecture of communication networks is

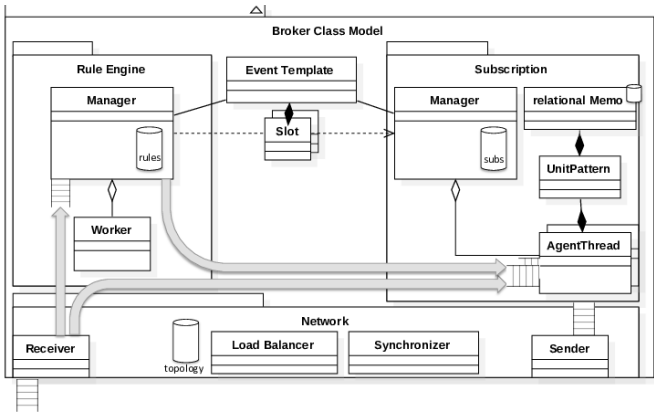

Fig. 2 Broker Class Model

simply pictured as in Fig. 1. All devices communicate to each other through broker nodes connecting to the mobile ad-hoc network with the same independent BSSID. We apply the Better Approach To Mobile Adhoc Networking (B.A.T.M.A.N.) protocol which independently implemented in Layer 2 for the basic network construction of our system.

In general, there are many ways to connect the device to IoT system via a tiny computer-on-module (e.g. Intel Edison). System clients (i.e. Human) remotely connect to the system over cloud server. They could be both subscribers and actors at the same time. For the machine devices, the communication could be performed in one-way or two-ways as well, but as a sensing device (i.e. publisher) or an actor. We might bridge broker to the local network router or directly connected to device by other available interfaces which could be wired (e.g. physical pin, USB serial port) or wireless (e.g. bluetooth).

3.2 Broker Design

Broker nodes play all main roles of the system: linking the rest components including subscribers, sensing devices, and actors, processing information flow from sensing device in response to subscriptions from system clients and forwarding to the final actors. It might be noted that some processed event might be further required by another subscription.

An overview of a broker node can be simply depicted from designed class models in Fig. 2, where the arrows denote information flows in the broker. There are three modules working together: network, rule-engine, and subscription.

- **Network Module** is responsible for cooperating with the other brokers for distributed deployment. It fundamentally contains four components: Receiver and Sender for basic functions of ad-hoc network communication, Synchronizer for synchronize all brokers to be in the same state, and Load balancer to balance the workloads for all nodes in the network. Additionally, we present some example idea of implementation for the last two components as follows:
  - **Synchronizer** should keep synchronizing subscriptions for all brokers in the system with very light control. All brokers might periodically broadcast the small hash value representing the holding subscriptions. When a hash value is not synchronized, only the older one will request for synchronization. For reducing network cost, a requested node will return just only some subscription based on some assumption, for instance, the subscription that was updated after the global synchronized time of the requester. However, a second request must be sent if the next hash-value comparison still results as not synchronized to ask for the rest subscriptions that were not sent from the last request.
  - **Load Balancer** might concern some cost model for distributing the workload. Basically, we should consider in three dimensions: processing complexity of the subscription, resources, and, finally, information flows. There are a number of publications for the first two dimensions but not for the last one. We suggest to determine probabilities of information-flow change over the network, and then decide which events to be processed by itself and which events to be forwarded for further processing by the next node. Fundamentally, there are two factors influences the flow change probability: network condition to the final actor and historical statistic of flow-in and flow-out. To decide the action, we must first consider the state of the node retrieved from node monitor unit is the node now overloading or not. If yes, it will then suggest to forward the information determined as causing no or small change of flow. However, the feedbacks of neighbors from feedback detector should be concern as well
- **Rule Engine** is responsible for testing tests the event flows with only simple constant condition defined in the subscription in primary step before forwarding to the subscription agents in the subscription module. However, some events might be forwarded from receiver to some subscription agents without passing through this module because of no simple condition required. In that case, broker will perform in the same way as stream processing. In our implementation, we apply a Java Expert System Shell: JESS [6] as a rule engine worker. Subscription agents will directly register new template as well as ask for setting a detecting rule for its simple conditions to the Rule Engine manager (RE manager). RE manger will simply handle duplicate request from different agents and then command the worker to do task. Afterwards, RE manager will cooperate with the subscription manger when subscriptions are revoked.
- **Subscription** is mainly responsible for processing the subscriptions. When a new subscription comes, Subscription manager (Subs manager) will initiate a new subscription agents to take a responsibility. An agents will parse the subscriptions and keep as a set of unit patterns (further explained in Section 4, and 5). It will notify the manager when the event corresponding to the subscription is fired. The event will be fired if all unit patterns are all correspondingly valid at the same time.

Table 1 General Form of Unit Pattern

Type	General Form
Simple	$subCE_A$
Time-range	$F(subCE_A)$ *within* T *from* $subCE_B$
Event-bound	$F(subCE_A)$ *between* $subCE_B$ *and* $subCE_C$

*F refers to selection, aggregation, negation

4. Combination Language

To support both continuous query and complex event detection, we define a new combination language referencing to the most expressive language in each area: TESLA, CQL. The advantage of the continuous query language (e.g. CQL) is that it can represent the relational operations between a set of windowed and structured tuples. Conversely, complex event detections handle detected event in the engine like rule-based engine or automaton engine. Owning to no need of database connection, the complex event can be detected faster and in parallel. Also, TESLA language can represent the condition in terms of sequence much more expressive than CQL.

4.1 TESLA

According to [3], a general structure of composite-event specification (i.e. rule) in TESLA language is represented as follow:

$$\begin{array}{ll} \textit{define} & CE(cAtt_1, ..., cAtt_n) \\ \textit{from} & Pattern(subCE_1, ..., subCE_j, ..., subCE_m) \\ & \quad = UP_1 \textit{ and } ... \textit{ and } UP_t \\ \textit{where} & wAtt_1 = f_1, ..., wAtt_k = f_k, ..., wAtt_n = f_n \\ & ; f_k := f(x_1^k, ..., x_p^k, .., x_q^k);\ x_p^k \in cAtt(\exists subCE_j) \\ \textit{consuming} & e_1, ..., e_i, ...e_h; e_i \in subCE, h \leq m \end{array}$$

Pattern in $from$ clause composes of a set of unit patterns (UPs) associated with *and* conjunction. There are three possible unit patterns represented in general form as shown in Table 1. A unit pattern has one base sub-event (i.e. $subCE_A$) with matching attribute-conditions. To define matching conditions, there are two types of operators: comparison, $[cmp\_op] \in \{>, <, \leq, \geq, =, ! =\}$, and parameterization, \$. The general form of sub-event with matching conditions in TESLA language is represented as: $\mathbf{subCE}_A((\boldsymbol{fAtt}\ [cmp\_op]\ value)^*|(\boldsymbol{pAtt}\ =\ \$varname)^*)$

4.2 Continuous Query Language (CQL)

Continuous Query Language (CQL) is based on SQL with additional constructs to support data stream processing. Similarly, the general form is basically composed of *select* clause, *where* clause, and *from* clause. Nevertheless, CQL further defines three groups operations to processing data streams by the input and output of the operation: stream-to-relation, relation-to-relation, relation-stream.

First, a stream-to-relation operation group is the operation group for windowing the continuous stream to keep in relational database. CQL defines three operations: (1) Time-based, S [*Range* T], (2) Tuple-based, S *Rows*[N], and (3) Partitioned, S [*Partition By* $A_1, ..., A_k$ *Rows* N]. Second, a relation-to-relation group includes all relational operations referenced in SQL query. Third, a stream-to-relation operation group defines the way to create a stream from the continuous changed database. CQL also defines three operations for this group: (1) Insert stream, inserted tuples, $Istream(R)$, (2) Delete stream, deleted tuples, $Dstream(R)$, and (3) Relation stream, all tuples, $Rstream(R)$.

Table 2 Unit Pattern

General Form:
[*Sel*] $SubCE_A$([*Comp*]) [Group by Attribs] [*Limit*]

Selection Policy[Sel]: $\phi(\equiv each)$ [*distinct*|*outer*]| $first$ | $last$

Comparison Condition[Comp]

Independent*	Dependent
$f(Attrib_i) \oplus constant$	$Attrib \oplus f(subCE_{x,x \neq A}.Attrib)$
$f(Attrib_i) \oplus f(Attrib_{j,j \neq i})]$	$Attrib = \$varname$

**Relational:* f function is aggregation
$\oplus$ includes all comparison operators defined in TESLA

Relational Limitation Policy[Limit]

Dependent	Independent
ϕ ($\equiv$ *Unbounded*)	*within* T *from* $subCE_{x,x \neq A}$
Range T	*between* $subCE_{x,x \neq A}$
Rows N	*and* $subCE_{y,y \neq \{A,X\}}$
Partition By Attribs	
Rows N	
Now	

4.3 Combination Language

A TESLA structure can cover CQL structure while CQL cannot due to consuming clause. A *select* clause in CQL is presented in *where* clause in TESLA while $from$ and *where* clauses are combined and presented in $from$ clause. The alias name for temporary table can be considered as event name in $define$ clause. So, we hold to TESLA structure.

At the same time, some operations are not directly support in the current TESLA especially for sliding window. Because TESLA is designed for detecting, there is no concept to limit the number of historical data. It might be noted that windowing operation can be performed in indirect way by generating database-change notification. Although inner-join can apply parameter operations (= \$), outer-join and grouping for aggregation is not available. Our new combination language complement valid patterns in $from$ clause of the TESLA language for the mentioned issues.

In our language , a unit pattern is composed of three parts: comparison condition, relational limitation policy, selection policy, and consumption policy (see Table 2). The process engine that applying this language must handle the information flow as streams of events. To simply explain the flow of processing, comparison policy tells which events should be memorized, relational limitation policy tells when to discard the memorized events, and, finally, selection policy tells which memorized events will be fired.

The comparison condition part defines the matching conditions of the base sub-event. It can be separated into two groups depending on whether it refers to the base sub-event

in the other unit patterns: independent and dependent. For independent group, the condition can further labeled as relational condition if any attributes of the base sub-event attached to aggregation function. In addition, grouping operation is allowed to define right after the comparison condition part as shown in Example 1. Notice that it is also included having operation in *where* clause.

Example: 1

Define	TempOver25
Where	$room\_id = T.room\_id$,
	$avgTemp = T.avgTemp$)
From	$Temperature(avg(val)$
	as $avgTemp > 25$) group by $room\_id$ as T

The relational limitation part defines the limitation policy of the relational database which keeping the matching base-sub-event. Similarly to comparison condition, the relational limitation policy can be also classified into two group by the relation between unit pattern: dependent, independent. All sliding windows in CQL are included in the independent class while event sequences (i.e. *between* and *within*) are classified as dependent class. This part can be omitted for the unbounded policy.

For the selection policy, in the same way as TESLA definition, there are three possible selection policies: each, first, last. Nevertheless, they also imply the relation-to-stream operations in CQL, Rstream, Dstream, Istream respectively, when combining with independent bounded limitation policy. This part can be omitted when representing *each* policy. In addition, the projection, *distinct*, and outer-join (i.e. alternative sub-event), *outer*, also represented here for only *each* policy.

To consume which unit pattern can be defined explicitly in *consuming* clause or implicitly defined. There are two cases for implicit consumption to avoid the run-out-of-memory problem in the broker: (1) the unit pattern that activates the other unit patterns (2) the unit pattern that has unbounded limitation policy.

To demonstrate, an example for converting CQL to our combination language is shown in Example 2.

Example: 2

Define	HRoverThreshold
Select	$Rstream(subject\_id,\ avg(val)$ as $avgHR)$
From	$HeartRate$ [Partition by $subject\_id$ Row 10]
Group by	$subject\_id$
Having	$avg(val) > 100$
	$\Downarrow$
Define	HRoverThreshold
Where	$subject\_id = HeartRate.subject\_id$,
	$avgHR = HeartRate.avgHR$)
From	$HeartRate(avg(val)$
	as $avgHR > 100$) group by $subject\_id$
	Partition by $subject\_id$ Row 10

5. Implementation

An IFP engine must at least support these following functionalities: (1) Subscription Registration, (2) Subscription Revocation, (3) Information Flow Processing. Our design IFP engine can achieve all functionalities by cooperating between Rule Engine module and Subscription module as explained below:

Table 3 Limitation Policy Summary

Policy	(1)	(2)	(3)	(4)	(5)
now	no	1	no	no	no
within	two	T	no	yes	no
between	two	∞	yes	yes	no
range	one	T	no	no	no
rows	one	N	no	no	no
partition	one	A_i,N	no	no	no
unbound	one	∞	no	no	yes

(1) **Subscription Registration** As primarily mentioned in the Section 3, Subscription Manger will initiate an Subscription Agent to handle all the rest processes. The agent will parse the subscription in form of event definition as explained in Section 4 for each parts as following procedures.

- *define:* subscription name (equals to event name), attributes (may including types) are extracted and sent to register as new template to the Rule Engine manager (RE manager).
- *where:* assignments are considered as selection clauses in the final steps of processing.
- *from:* patterns are split into at least one unit patterns by *and* conjunction. Then, each unit pattern further compile each parts. First, it can determine the following issues from limitation policy: (1) how many relational databases are needed, zero, one, or two? (2) If (1) yes, how does it limit the database? (3) does it have to wait for activator? (4) does it have to wait for finish event? (5) is it forced to be consumed? (see summary in Table 3). For unit patterns with activator, the first memory also keep the rowid of the activator for the checking corresponding at last. Similarly, unit patterns with finish event are required to keeps the rowid of the finish event as well. Condition part is split with *and* conjunction and then classify into four groups: constant, relation, dependent, and parameter as explained in Section 4. The constant condition set from each unit pattern is submitted to the RE manager to generate a detecting rule.
- *consuming:* explicitly-defined consumed unit patterns are set.

(2) **Subscription Revocation** This functionality is performed by Subscription Manager who keeping all agents information. It will clear all relevant temporary databases before kill the agent (i.e. interrupt while loop of Thread). Then, it will notify the rule-engine manger to remove the related rules that has no other agents subscribing.

(3) **Information Flow Processing** Fundamentally, there are three steps to process complex event from information flows respective to conditions defined in the subscription as summarized in 1: constant step,

intra-relation step, and inter-relation step. The first step is execute at the worker of rule engine module while the rest are done by the subscription agents. Note that, the first step can be passed through in case that there is no constant condition. The matching event will added to each unit pattern according to its database condition. If it is distinct, the previous one will be replaced with the new one. The over-limit data will deleted. In case of $first$ selection policy with independent limitation policy, the deleted data is recorded. Then, second step, agents will test relation conditions with grouping attributes in the result database. If pass the second step, unit pattern will be considered as valid. Note that, the results will become invalid if there are some corresponding events remains in the memory of the unit pattern with negation operator. If all results from unit patterns are valid at the same time, agents will join those results using sql query with dependent condition in *where* clause, parameter condition in *joinon* clause, and where assignment in *select* clause. Joining is inner as default, but, *outer* is used to deprecate *outerjoin* in this step. Finally, the tuple results will be converted to out event stream.

6. Preliminary Implementation Test

To confirm our combination design, we implement a simple engine on Java using Jess as rule engine at the first state before forwards the matched results to relational unit connecting to Sqlite database. Running tests are applied with three subscriptions (see. Table 4). We define the subscriptions by concerning the advantage points of the data-stream approach against rule-based approach.

The first example is the superior situation for rule-based engine where no relational computation among event. In addition, the temporary copies of event in the engine is supposed to be small because of negation operator. While the second one represent the disadvantage point of the rule-based approach where there is only one relational operation. For the last case, we balance the advantage between relational operation and comparison operation to represent the best appropriate case for our proposed approach.

Input streams include three types of data: Heart Rate, Movement, and Temperature. We use the real-collected Heart Rate data and Movement data from three subjects and three Temperature data from available example sources on the Internet.

The results in Fig. 3 show time delays to process in nanoseconds accumulating over the number of events in the flows. We could observe that our combination approach process data relatively faster than using relational database only in the first case while remaining the same rate for the last two cases. However, our preliminary tests are conducted on only in thousands of data in a minute. In the future work, we are going to implement the complete engine and evaluate dealing with much bigger amount and more various data by more complex subscriptions.

Algorithm 1 IFP for Multi-purpose IoT system

procedure 3-STEPS CEP
  (1) Constant Step
  Match Engine:
  if *event* $\mapsto$ *constant_conditions* **then**
    Send to subscribing SubsAgents
  (2) Intra-relation Step
  Subscription Agent:
  event main sub_event_A:
  if *has_relation* **then**
    if has *activator* **then**
      Add to relation with *bid*
    else
      Add to relation
    if *relation_condition* **then**
      if no $finish\_event$ **then** *result* $\leftarrow$ *relation*
  else
    if no $finish\_event$ **then**
      result $\leftarrow$ *event*
    else
      temp $\leftarrow$ *event*
  event as activate sub_event_B:
  if UP_B has relation **then**
    bid $\leftarrow$ *rowid* of event in UP_B
  else
    bid $\leftarrow$ 0
  event as finish sub_event_B or sub_event_C:
  if *has_relation* **then**
    select $\leftarrow$ Selection Policy
    where $\leftarrow$ *relationcondition*, *event*
    query first relation with *select*, *where*
    if *query* **then**
      Add to second relation with *cid*
    result $\leftarrow$ *secondrelation*
  else
    if correspond to *event* **then**
      result $\leftarrow$ *temp*
  (3) Inter-relation Step
  if all *result* **then**
    select $\leftarrow$ assignment in *where*
    where $\leftarrow$ *dependent_condition*
    $from$ $\leftarrow$ join all results on *parameter_condition*
    query with $select, where, from$
    if query **then**
      return *query* as event stream

Table 4 Example Subscription for Evaluation

Example 1:	
Define	**HighHRNoMove**(subjectId:STRING,hr:INTEGER)
Where	$subjectId = HeartRate.subjectId,$ $hr = HeartRate.hr$
From	$HeartRate(subjectId = \$id\ and\ hr > 110)\ AND$ $not\ Movement(subjectId = \$id\ and\ val = 1)$ $within\ 1\ min\ from\ HeartRate$
Example 2:	
Define	**MonitorAvgHR**(subjectId:STRING,avgHR:FLOAT)
Where	$avgTemp = Temp.avg(val)$
From	$Temp()range2mins$
Example 3:	
Define	**AvgTempInMeetingRoom**(avgTemp:FLOAT)
Where	$avgTemp = Temp.avg(val)$
From	$Temp(location ==' meetingRoom')range2mins$

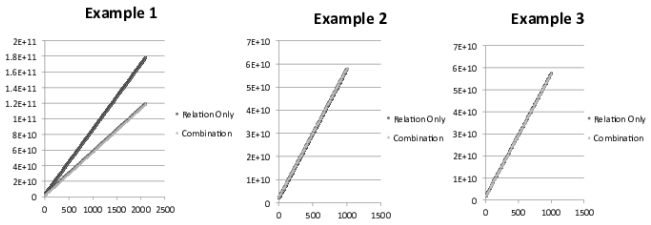

Fig. 3 Experimental Results

7. Conclusion

In this paper, we concern the co-existing of various kind of application needs especially in multi-purpose IoT system. Until now, there are two ways almost separately works covered as Data-Stream processing and Complex Event processing. The former one operates stream of data in relational database while the latter operates as an independent event. Each approach has its own advantage points: relational processing, special-event detection, respectively. Now, they still cannot work together completely. Some operations are not support for each other. We consider logical reasons behind them and propose a new definition that allows users to define their needs more expressive in one language. Also, we suggest the way to implement the combination of relational database and rule-based engine for supporting our combination languages in distributed-deployment architecture. Finally, we simply implement real engine according to our design and perform preliminary test on some significant examples of subscription as well as comparing our combination approach to relation-stream approach in terms of processing time in each example scenario.

References

[1] Arasu, A., Babu, S. and Widom, J.: The CQL Continuous Query Language: Semantic Foundations and Query Execution, *The VLDB Journal*, Vol. 15, No. 2, pp. 121–142 (2006).

[2] Cugola, G. and Margara, A.: RACED: An Adaptive Middleware for Complex Event Detection, *Proceedings of the 8th International Workshop on Adaptive and Reflective MIddleware*, ARM '09, NY, USA, ACM, pp. 5:1–5:6 (2009).

[3] Cugola, G. and Margara, A.: TESLA: A Formally Defined Event Specification Language, *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10, NY, USA, ACM, pp. 50–61 (2010).

[4] Cugola, G. and Margara, A.: Complex Event Processing with T-REX, *J. Syst. Softw.*, Vol. 85, No. 8, pp. 1709–1728 (online), DOI: 10.1016/j.jss.2012.03.056 (2012).

[5] Cugola, G. and Margara, A.: Processing Flows of Information: From Data Stream to Complex Event Processing, *ACM Comput. Surv.*, Vol. 44, No. 3, pp. 15:1–15:62 (2012).

[6] Friedman-Hill, E.: Jess, The Rule Engine for the Java Platform, `http://herzberg.ca.sandia.gov` (2007).

[7] Li, G. and Jacobsen, H.-A.: Composite Subscriptions in Content-based Publish/Subscribe Systems, *Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware*, Middleware '05, NY, USA, Springer-Verlag New York, Inc., pp. 249–269 (2005).

[8] Li, G., Muthusamy, V. and Jacobsen, H.-A.: *Middleware 2008: ACM/IFIP/USENIX 9th International Middleware Conference Leuven, Belgium, December 1-5, 2008 Proceedings*, chapter Adaptive Content-Based Routing in General Overlay Topologies, pp. 1–21, Springer Berlin Heidelberg (2008).

[9] Liang, C.-J. M., Chen, K., Priyantha, N. B., Liu, J. and Zhao, F.: RushNet: Practical Traffic Prioritization for Saturated Wireless Sensor Networks, *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, SenSys '14, New York, NY, USA, ACM, pp. 105–118 (2014).

[10] Madden, S. R., Franklin, M. J., Hellerstein, J. M. and Hong, W.: TinyDB: An Acquisitional Query Processing System for Sensor Networks, *ACM Trans. Database Syst.*, Vol. 30, No. 1, pp. 122–173 (online), DOI: 10.1145/1061318.1061322 (2005).

[11] Pietzuch, P. R., Shand, B. and Bacon, J.: A Framework for Event Composition in Distributed Systems, *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, Middleware '03, New York, NY, USA, Springer-Verlag New York, Inc., pp. 62–82 (2003).

[12] TIBCO: StreamBase, `http://www.streambase.com` (2003).