ReCEPtor: Sensing Complex Events in Data Streams for Service-Oriented Architectures

Mingzhu Wei, Ismail Ari, Jun Li, Mohamed Dekhil
Digital Printing and Imaging Laboratory
HP Laboratories Palo Alto
HPL-2007-176
November 2, 2007*

data stream, complex event processing, SOA, query plan, EPL

All mission-critical applications read or generate raw data streams and require real-time processing of these streams to collect statistics, control flow and detect abnormal patterns. A trend which has gained strong momentum in different industry sectors is the use of Complex Event Processing (CEP) over streams for all critical business processes, thus pushing its span beyond military and financial applications. A parallel trend has been the re-architecting of existing business processes with Service Oriented Architecture (SOA) principals to provide integration and interoperability. This requires use of middleware that incorporates web services, Business Process Management (BPM) systems or an Enterprise Service Bus (ESB), and in some cases Business Rule Engines (BRE). There is a gap between the new generation of business processes which desperately need CEP and the proposed CEP engines that were not built with SOA in mind. These engines either don't support the flexible, dynamic and distributed business applications deployed in SOA or they try to merge the middleware with the CEP engine as one big proprietary package.

This paper describes the design and implementation of our CEP engine called ReCEPtor, which can sense complex events in data streams in real-time. Our modular architecture describes how to integrate ReCEPtor with different business applications in different platforms either directly or by using an off-the-shelf BPM system and a platform adapter. We address issues related to CEP system flexibility, interoperability, scalability and performance in this paper. We also discuss novel concepts such as sequence operators, query plan adaptation, and scheduling for progressive flow. We can already process ~15 requests/second through the 3-Tiers (web services, orchestration, and database) of our prototype end-to-end and without tuning and we are currently working on achieving higher speeds.

Internal Accession Date Only

Approved for External Publication

© Copyright 2007 Hewlett-Packard Development Company, L.P.

ReCEPtor: Sensing Complex Events in Data Streams for Service-Oriented Architectures

Mingzhu Wei[†], Ismail Ari, Jun Li, Mohamed Dekhil

Hewlett-Packard Laboratories

Worcester Polytechnic Institute[†]

Abstract

All mission-critical applications read or generate raw data streams and require real-time processing of these streams to collect statistics, control flow and detect abnormal patterns. A trend which has gained strong momentum in different industry sectors is the use of Complex Event Processing (CEP) over streams for all critical business processes, thus pushing its span beyond military and financial applications. A parallel trend has been the re-architecting of existing business processes with Service Oriented Architecture (SOA) principals to provide integration and interoperability. This requires use of middleware that incorporates web services, Business Process Management (BPM) systems or an Enterprise Service Bus (ESB), and in some cases Business Rule Engines (BRE). There is a gap between the new generation of business processes which desperately need CEP and the proposed CEP engines that were not built with SOA in mind. These engines either don't support the flexible, dynamic and distributed business applications deployed in SOA or they try to merge the middleware with the CEP engine as one big proprietary package.

This paper describes the design and implementation of our CEP engine called ReCEPtor, which can sense complex events in data streams in real-time. Our modular architecture describes how to integrate ReCEPtor with different business applications in different platforms either directly or by using an off-the-shelf BPM system and a platform adapter. We address issues related to CEP system flexibility, interoperability, scalability and performance in this paper. We also discuss novel concepts such as sequence operators, query plan adaptation, and scheduling for progressive flow. We can already process ~15 requests/second through the 3-Tiers (web services, orchestration, and database) of our prototype end-to-end and without tuning and we are currently working on achieving higher speeds.

1 Introduction

A data stream is a real-time, continuous, unbounded, possibly bursty and time-varying sequence of data elements. These data elements could be either relational tuples [41] as seen in traditional databases or structured Extensible Markup Language (XML) elements [57]. Many applications generate data streams including sensor networks [45,23,12], financial tickers [23], news feeds [39,61], online auctions [53,46], web click-stream [20,14], network-system-traffic monitors [7,3], and supply chain systems with Radio Frequency Identification (RFID) tracking [26]. Organizations running these systems and applications need continuous monitoring and real-time processing of data streams to complete mission-critical tasks and to survive their competition.

It is quite challenging to continuously analyze and merge raw data streams so that the results make actionable and operational sense. This is the goal of CEP or Event Stream Processing (ESP), which has recently gained strong momentum both in the academia and in the industry. Specifically, retail and financial industries present many business-critical use case scenarios, which can be categorized as complex events. We present two running examples in this paper related to these two industries: real-time personalized retail campaign management and detection of high-value financial customers over multi-channels of a bank. To detect such events, retailers need to monitor inventory levels and banks need to know immediately when a valuable customer contacts them for an offer. Lots of information is buried inside raw data streams, but the real actionable results can be generated via CEP. One can try to use raw streams directly to make business sense, but this is both impractical due to the need for manual parsing of huge data volumes and insufficient as higher payoffs are usually gained from complex scenarios.

SOA and Web Services (WS-*) are being prevalently used today to address integration and interoperability problems of distributed Information Technology (IT) services and applications. New services are being developed and deployed and existing components are being wrapped as web services for reuse. A BPM system or an ESB lies at the core of SOA to provide message routing and choreography among these open services, so that the dynamic, distributed applications can be implemented and can make progress consistently. A CEP engine can be an invaluable component inside a SOA by sitting on top of raw data streams and providing simple and complex event sensing features to these applications and services. Proposed CEP engines focus on new language semantics, performance, and robustness aspects of design putting less effort on flexibility, interoperability and reusability of these engines. We find these aspects to be especially critical for wide adoption of CEP in service-based or event-driven architectures (EDA) [44].

To sense complex events in data streams address mentioned challenges and implement novel use cases, we built a CEP engine called *ReCEPtor* on top of a Data Stream Management System (DSMS) called CAPE. Our goals include detecting complex events in real-time, making CEP engine flexible and reusable for different applications that react to complex events, supporting online query registrations, exploiting sharing among multiple queries, and finally investigating performance and scalability issues that arise with high-speed streams. We used an algebraic approach instead of automaton for generating the stream operators and the query plan. We find our approach to be inline with the novel dataflow paradigm adopted by other recent projects.

We integrated our CEP engine with web-based and desktop proof-of-concept applications that we designed for demonstration purposes. We show that ReCEPtor can serve different applications for different industries. Specifically, we investigated the retail and retail banking environments. Using CEP creates new service opportunities for these industries including multi-channel customer tracking and prospect detection, real-time micro-campaigns, demand shaping, arbitrage and fraud detection, business process activity monitoring [25], and finally exception handling [40]. Finally, we also demonstrate platform independence by combining .NET and Java implementations using Common Object Request Broker Architecture (CORBA). As minor contributions in this paper, we discuss a list of complex event scenarios with real motivations in the industry and provide a comprehensive discussion for open and interesting research problems.

The next section summarizes basic concepts in data stream systems and compares them to Database Management Systems (DBMS). We describe the overall CEP architecture in Section 3 and CEP engine design details in Section 4. Section 5 gives preliminary performance results, implementation details of the current prototype and shows proof-of-concept applications. Section 6 compares and contrasts our work with other related work. Section 7 concludes the paper and summarizes interesting future work.

2 Background

In this section, we introduce basic concepts in data stream processing research. First, we compare traditional DBMS with Data Stream Management Systems (DSMS) and list complex event characteristics to emphasize their differences. Next, we overview stream query languages and introduce basic stream operators, which are used to write continuous queries submitted to DSMS. Finally, we describe the sliding-window stream join operator, which can be used to detect complex patterns.

2.1 DSMS vs. DBMS

Data stream management systems [43,19,1,46] operate in a different mode than any other known databases and database extensions including Active databases [42], or Operational Data Store [31]. Figure 1 illustrates some of these differences. First, in DSMS the data from streams are processed or queried on-the-fly before they are (optionally) persisted into the database. Second, a data stream is unbounded (it keeps coming) and therefore queries are registered with the DSMS once and stay

there (forever or for a while) instead of being one-time request-responses. Finally, queries can be window-based which makes them suitable for quick detection of violations and abnormal sequences.

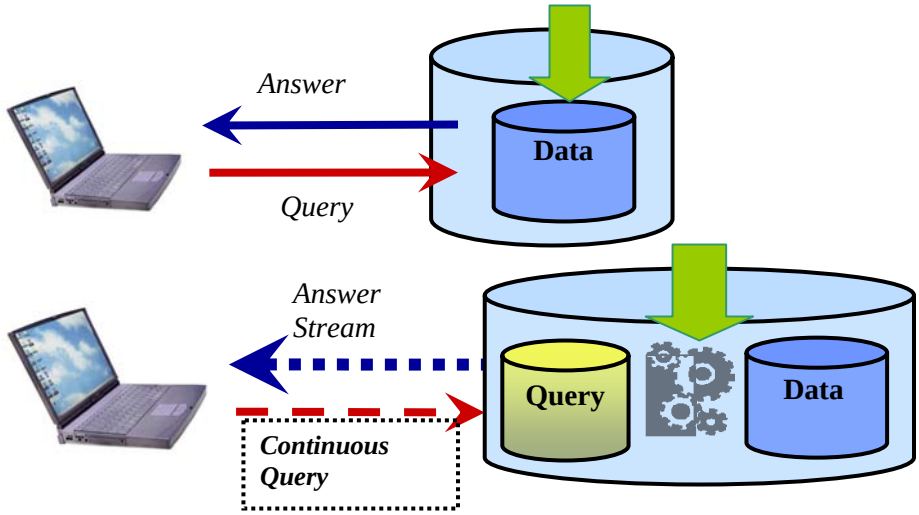


Figure 1: DBMS are characterized by requests followed by synchronous responses. In DSMS a continuous query is registered once and the responses are received as a stream asynchronously.

To emphasize the unique use cases for DSMS and CEP, we list three main categories of complex events that we frequently observe. First class is threshold-violations or filtered queries [61]. It is often the case that stream readings are continuously checked and compared against a pre-defined value to see whether a Boolean predicate (=,<, ≥) is satisfied. The second class constitutes of temporal or spatial [60] event patterns such as sequences of correlated events [48,26]. For example, when a credit card is used in the East and West coasts (spatial) within a few minutes (temporal), then we would suspect of a fraud. Finally, we observe a third class of complex events which contains non-occurrence of events. A "shoplifting" activity in a retail store of an RFID-tagged item contains the "non-occurrence of a Point of Sale (POS) reading" [26] after the shelf and before the gate readings. Note that many complex events will also entail a sliding window and a session (begin-end) constraint. Different complex events can be cast into these three categories.

2.2 Stream Query Languages

Stream query languages [23,7,26,58,52] such as Continuous Query Language (CQL) in STREAM system [7,20] use similar syntax to Structured Query Language (SQL) of DBMS, but extend it with operators required for continuous stream processing. These languages are used to write the query statements that will be registered with a DSMS. A continuous query statement commonly carries the following components:

```
SELECT    …Output data
FROM      …Stream
[WHERE]   …Predicate
[WITHIN]  …Window size
[EVERY]   …Window slide
```

SELECT clause specifies the format of the output data, it can use a projection operator to project a few or all (*) columns from an input stream as in A.EventType to the output. Similar to SQL aggregations such as Count, Sum, Min, Max, Avg can also be used with SELECT. FROM clause specifies the source data stream(s) that will be used to make the selections. Since a data stream is unbounded, we cannot wait forever to compute the query results. Therefore, a "window" constraint

is proposed to approximate and process a finite subset of the streams, which is specified by the WITHIN clause. There are different options for windows management: time-based (last 10 minutes), tuple-based (last 1000 tuples), or value-based (values 2t, 2t+5 in an increasing sequence) [23,14]. Since queries execute continuously, we need to keep advancing the window to process newly arriving data. EVERY clause defines how far to move a sliding window or with which interval to refresh the results. The time-based window will usually be periodic and the others non-periodic [37]. In addition, periodic windows can slide in an overlapping, tumbling or hopping fashion [37,16]. WITHIN and EVERY clauses are respectively called RANGE and SLIDE in CQL [7]. Other temporal clauses (AFTER, AT, UNTIL, SINCE [3], DISTINCT [28]) and set clauses (EXCEPT [16]) were also suggested in the literature. Optionally, an operator can PUBLISH [61] or INSERT [7] its output stream to be reused by other queries registered in the system to increase efficiency. We discuss these details in Section 4.

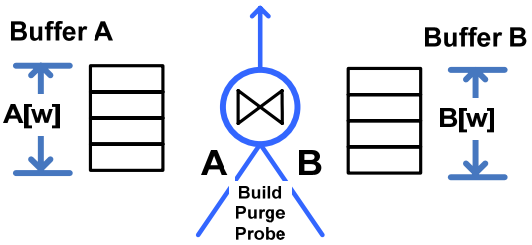
Consider the following example where retail shoppers with loyalty cards visit a store kiosk to obtain personalized coupons based on their shopping history. To monitor the success of its real-time promotional campaigns and tune algorithms, the retailer wants to track how many personalized coupons were viewed within the last 10 minutes and refresh the results every minute. We can register this query using a CQL statement as below:

```
SELECT Count(*)
FROM   Coupons
WHERE  Coupons.State = "Viewed"
WITHIN 10 minutes
EVERY  1 minute
```

When stream query languages are extended further with operators to support complex event processing, they are sometimes called Event Processing Language (EPL) [5,26,33,61].

2.3 Time-based Sliding Window Join

We use join operators (indicated by ⋈) to implement event sequence queries with sliding windows [58,34]. Consider the query below, which finds the pair of tuples with the same identifiers in streams A and B within one hour. A binary-join operator could be used in the query plan to implement this query shown in Figure 2. This operator maintains both A and B states for the two sides to hold the arriving tuples. Each new stream A tuple should join with those tuples in B's state whose time difference is less than 1 hour. Therefore, we need to run three basic operations build, purge and probe for each new arrival.




```
SELECT A.Name, B.Amount
FROM   A, B
WHERE  A.Id = B.Id
WITHIN 1 hour
```

Figure 2: Sliding window-based join

Build means inserting a new tuple say from stream A to the state of A. Purge means comparing the timestamp of this new A tuple with the timestamps of all the tuples in B's state to purse out-of-date (i.e. out-of time-window) tuples. After the out-of-date B tuples are purged A tuple will probe B's remaining state to get the matching results for the Id equivalence. Similarly, new arrivals from stream B will follow the same build, purge and prove operations on A's state. Note that purge has

to be executed before probing to avoid reprocessing of B's state for out-of-date tuples on new arrivals and reduce computational overheads. While efficient, this method can lead to incomplete event retrieval in the existence of out-of-order arrivals in data streams [34].

3 CEP Architecture

Figure 3 shows our CEP architecture and integration with business-level applications. We identify four layers of interest (from bottom to top): 1- Raw data streams, 2- CEP engine, 3- Middleware, and 4- Applications such as visualization and monitoring tools. It is also possible to view this architecture in 3-Tiers where the web services constitute Tier-1, the orchestration or the application server constitutes Tier-2, and the event repository in the database constitutes Tier-3. CEP engine could belong to either Tier-1 ("CEP As A Service") to be directly used by applications or Tier-2 (CEP through BPM) in a SOA to be coordinated with other services before being used by applications. We show the second option here.

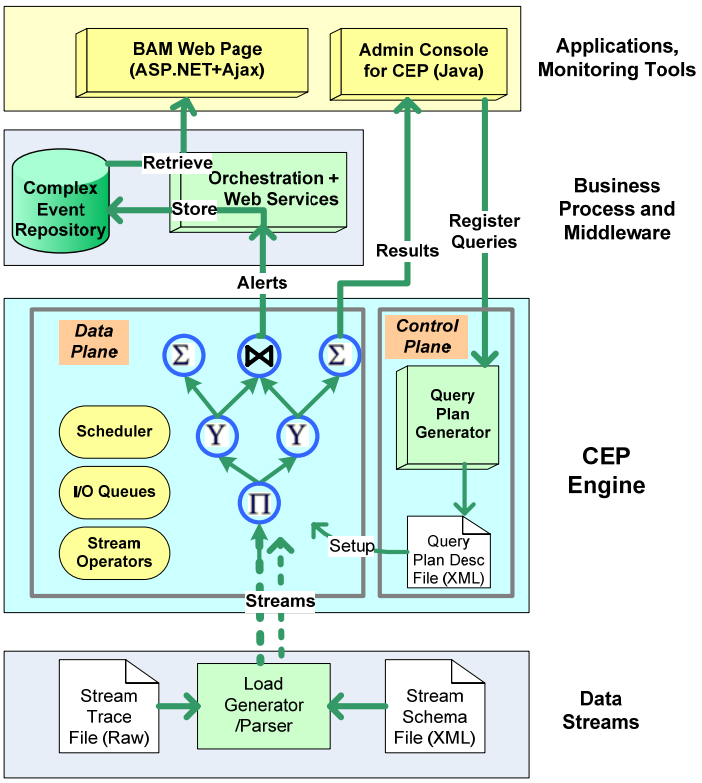


Figure 3: Complex Event Processing (CEP) for Service-Oriented Architectures.

3.1 Raw Data Streams

Continuous data streams are the inputs that activate our CEP engine. Retail and financial industries deliver a big variety of raw data streams including monetary transactions, stock quotes, Customer Relationship Management (CRM) system records, online click-streams, and kiosk interactions. Our system can either read streams from real-applications through different I/O communication mechanisms like sockets or generate streams internally for testing and emulation purposes. The stream/load generator illustrated in Figure 3 schedules tuples either by using a historical trace file or relying on a schema and a distribution function like Poisson.

3.2 CEP Engine Overview

Our CEP engine has a control plane and a data plane as shown in Figure 3. Similar architectures can be found traditional and data snooping routers [8]. The control plane accepts multiple continuous queries and places them in a running query plan by exploiting sharing among the queries. The data-

plane is where all the processing takes place on raw data streams that are passing through. The output of the data plane is either statistics (Count, Avg, etc.) or complex events. Continuous queries written using a special EPL can be registered with the CEP engine through a management console. Optionally, a plan can be instantiated from a previously designed "query plan template" file for an application by only updating required parameters. There could also be an already running plan and the new query could be added to it as described in Section 4.3. The scheduler inside the engine will schedule tuples from streams to relevant operators' I/O queues. It decides the operator execution sequence. Stream operators such as Select, Join, and Aggregators are operationally similar to their SQL counterparts, but have completely different internals due to their time-based capabilities. For example, operators maintain states as described in Section 2.3, retain them in a Hashtable or a Queue structure, and sort the tuples based on the arrival timestamps. We design new operators (described in Section 4.3) to detect complex event patterns while also promoting sharing among similar concurrent queries.

Given the registered queries, the CEP engine has to generate a query plan, which is a list of required stream operators, their attributes and connections among them. To enable reuse of operators for sub-streams and increase plan flexibility, we need other intermediate operators. We define COPY (Y) and SPLIT (Π) operators in Section 4.3.2 and describe how to utilize them to improve the efficiency and flexibility of query plans with multiple queries. The query results can be sent to applications directly via TCP/IP sockets or through the middleware. Design and implementation details of CEP engine are given in Section 4.

3.3 Middleware and Business Processes

Middleware glues the CEP engine to the business application layer through web services and other bridges including CORBA. This level includes a business process also called a workflow that implements the distributed application logic. This process stores events into an event repository until they are retrieved by applications such as Business Activity Monitoring (BAM) tools. BAM is an application that can probe into business processes to monitor certain statistics [25]. With the help of CEP, BAM can also be used to observe occurrences of interesting events.

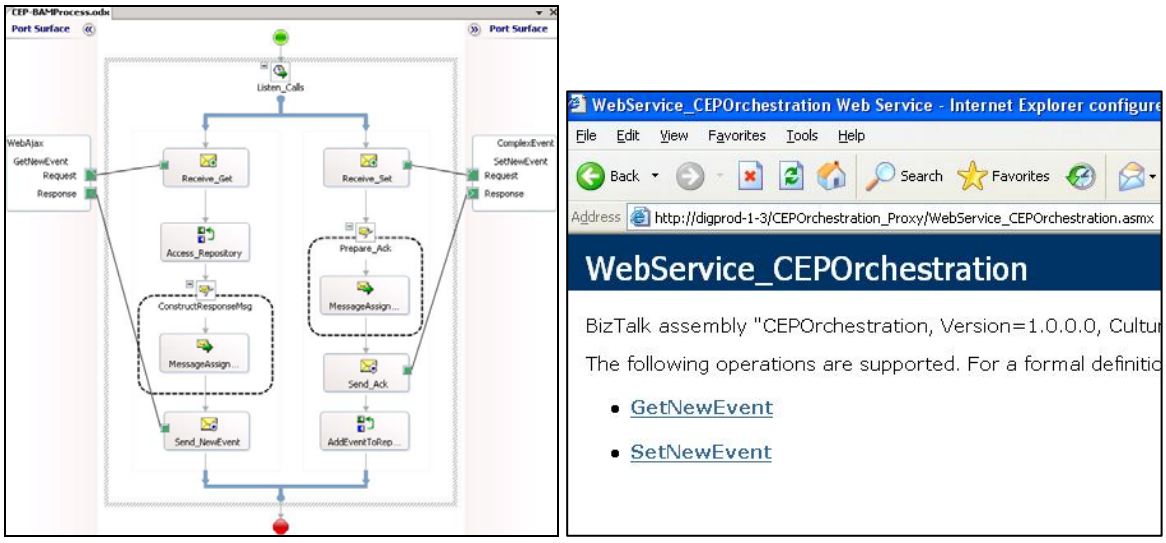


Figure 4: (a) Orchestration and (b) its published web services that integrate CEP to applications.

Our current workflow represents drastically simplified application logic and provides just two functionalities. One is to listen to query output from the CEP engine and the other is to listen to the requests from the application level. We implement it as an orchestration inside a BPM system by Microsoft called Biztalk. Figure 4 shows both the workflow and the methods published as web services. *SetNewEvent* method places new events it receives from the CEP engine into the complex

event repository. *GetNewEvent* listens to the application side and checks whether there are any new events in the repository that have not been fetched before to return these to the querying applications. The workflow sets a flag for new events and resets them after they're fetched. Complex event messages are sent from CEP to the workflow via Simple Object Access Protocol (SOAP). We will extend this demo workflow to support industrial strength use cases.

These simple functions could be implemented as web services directly on top of the database. However, this would limit the extensibility and flexibility of our system. For example, if we want to add a rule where "new events that have not been consumed by the manager's web portal within two minutes, should be sent to the manager via email or a short message", then we would need to (1) use a business rule engine (BRE), (2) be able to integrate with the email and SMS systems, and (3) understand business-level semantics to alert the manager about a "fraud" or "sales opportunity". These capabilities are beyond what a CEP engine core could and should provide.

It is crucial to note that we did not insert a BRE inside the CEP engine in our architecture, which differentiates our architecture from related work [3,5] and other pub/sub based CEP systems. We choose this approach because complex events usually trigger business-level responses determined by complex policies and require Enterprise Application Integration (EAI). In our architecture, we can easily build and deploy rules inside the BRE and access these rule-sets from the workflow system shown in Figure 4. The second reason for keeping BRE out of CEP engine is to assure separation of roles for scalability as rule engines can potentially serve thousands of rules for different applications inside the enterprise SOA.

Figure 5 illustrates different options for CEP-Middleware interaction. In addition to BPM system where the workflows are implemented, CEP can forward complex events directly to a Notification Service (NS) or the BRE. NS (e.g. by Microsoft) is publish-subscribe system [44] where incoming events (i.e. publications) are matched to previous subscriptions [4] and the results are sent to subscribers as notifications. NS can accept simple or complex events. In the context of BRE events can represent certain conditions (facts) that have occurred and the rule describes what actions to take based on the given condition. This is also called Event-Condition-Action (ECA) [33,42,3]. Different systems can be weaved together via enhanced messaging middleware sometimes called the Enterprise Service Bus (ESB) [21]. In Figure 5 the thicknesses of arrows symbolize the reduction of data volume as filters, aggregations, correlations take place within each layer to reach higher semantics: raw data→simple events→aggregate and complex events→business situations.

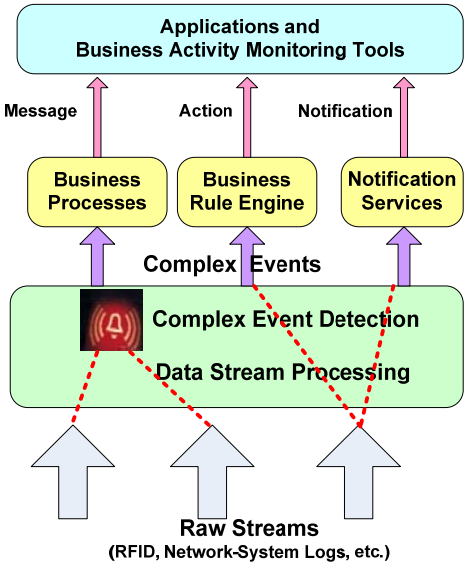


Figure 5: Integration of CEP engine with different event-driven systems.

3.4 Business Applications

Our goal in this paper is to design a flexible CEP engine that can easily be integrated with different applications such as BAM tools, web-based consoles, or pass-through administration consoles. We show some proof-of-concept applications in Section 5. In our vision, business applications register fraud-, statistics-, or prospect-related queries and CEP detects simple and complex events, also called situations [48]. Situations such as "shoplifting" in a retail store or a "high-value customer arrival" at a bank branch require instant-detection and timely-aggregation of multiple simple events. In response, the managers can either take manual actions or automate their responses by setting rules or policies such as setting a door alarm or alerting security via mobile all orchestrated through the BPM system.

4 ReCEPtor Design Details

This section describes the design and implementation of CEP engine. We first list the key design features and components in Section 4.1. We summarize capabilities of CAPE DSMS system on top of which ReCEPtor was built in Section 4.2. We describe the implementation of the sequence operator, query plan and scheduler in Section 4.3.

4.1 Feature Summary

ReCEPtor has the following features:

Detect complex patterns: We designed a new algebraic SEQUENCE operator to detect sequences of events. Non-occurrence of events can also be cast as a sequence detection problem where the missing event is represented by the occurrence of either a timeout or another out-of-sequence event which violates the expected order. We focus on detecting sequences in this paper.

Automated query plan generation: In CAPE system, which is currently used for research purposes, query plans are written manually and loaded into the system. In the practical scenarios, people who issue the queries may have little or no knowledge about a query plan and how it is organized. To address this issue, we worked on automated query plan generation for multiple continuous queries. The administrative consoles described in Section 5 are used to write these queries with basic EPL statements and to register these queries with the CEP engine. Currently, multiple queries are registered before the engine is started and more work needs to be done in the control plane to support online query registration.

Plan optimization: We support multiple queries in our CEP engine. Therefore, it is possible to have queries with common operators among the queries. We describe a plan generation method that reduces CPU and memory resource usage by improving sharing.

Interaction with application level: We need to provide a user-friendly interface to allow application development on top of CEP and to facilitate seamless business integration. As proofs-of-concept for interoperability we developed web-based and console-based applications.

4.2 Continuous Adaptive Processing Engine (CAPE)

Complex event processing can be considered a special type of stream processing, where the output has higher-level semantics than the output of basic stream operators. Building a CEP engine from scratch would require a lot of work, therefore we decided to build our CEP engine on top of CAPE, which is a DSMS developed by Worcester Polytechnic Institute (WPI) [46,38]. CAPE accepts streams with fixed relational schema and there is ongoing work to leverage its features to support XML streams in a related system called Raindrop [51].

CAPE system emphasizes the following novel features: (1) *Intra-operator adaptivity*: "As queries are registered into or removed from the query engine, the computing resources available for processing an individual operator may vary greatly" [46]. CAPE can exploit metadata knowledge in the data streams to reduce resource usage and improve execution efficiency of operators. (2) *Plan-*

level adaptivity: A query plan might also become sub-optimal at run-time. CAPE supports online query re-optimizations. (3) *System-level adaptivity:* CAPE supports adaptive scheduling query plan distribution among multiple machines for load balancing. Our current CEP engine does not utilize these features yet; therefore we skip the details [46,38] for brevity. We plan to extend our work to support distributed operation in the future to address large-scale, high-speed processing issues [14,60].

4.3 Sequence Operator, Plan Optimization and Progressive Scheduling

First, we describe the design of SEQUENCE operator, which uses a Semi-Join for detection of strictly-ordered event sequences. Next, we discuss the plan generation for multiple queries. We added two new operators: *Split* operator avoids useless tuple processing along a query path and the *Copy* operator feeds shared sub-streams to different operators. Finally, we describe a new bottom-up scheduling algorithm.

4.3.1 Sequence Operator

Sequence is about detecting the temporally or spatially correlated events. There are two ways to implement the SEQUENCE operator; one is to use a Non-Deterministic Finite-State Automaton (NFA) and implement sequence scan and construction over a stack [58,34] and the other is to use an algebraic approach where the comparison of timestamps is regarded as the join condition among input streams. Stack-based implementation can generate results directly without intermediate results. However, the stack-based automaton is relatively tied or fixed to the sequence pattern. In the case of multiple sequence queries registered in the CEP engine some may have common parts and algebraic approach provides the ability to exploit sharing among such queries.

Algebraic approach also enables long and flexible sequence patterns. We use the Semi-join operator illustrated in Figure 6 and use a retail scenario to explain its operation. Note the two state buffers located on both sides of the join operator. For example, a retailer can track the list of customers who have shown interest in certain offers or "viewed offers within 20 minutes after they were created" for real-time micro-campaigns and demand shaping. Each coupon "viewed" event that could be streamed from the kiosk will probe all the coupon "created" events in A's state that were streamed from the offer server to generate a strictly ordered sequence. With this approach we will have various choices to generate even longer sequences as will be described next. Another advantage of algebraic approach is that it is easy to apply equivalence predicates (e.g. for id comparison) on the sequence, which is commonly required to detect correlations. SASE uses partition-based stack construction [26] to apply predicates, which is only limited to one comparison predicate. In real applications multiple comparisons might be needed. For instance, detecting a high-value customer who uses multiple channels involves the comparison of both the customer and channel ids. In our case, it is easy to apply one or more predicates to the join operator.

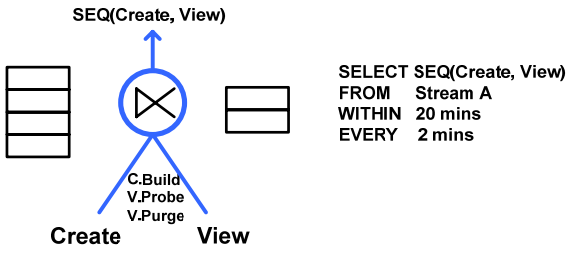


Figure 6: Algebraic approach to implement Sequence operator

Figure 7 shows different options to detect a long event sequence such as A→B→C→D→E. We can use binary (2-way) joins all the way through [41], combine a 3-way join with a 2-way join, or use an arbitrary multi-way join that represents the complete sequence. However, there is always a trade-off between flexibility and efficiency or performance. Pipelined execution of binary joins [28]

provides more opportunities for intermediate result sharing for multiple sequence queries. However, whether it is 2-way or n-way joins, the algebraic approach provides flexible and stackable sequence operator implementation for multiple continuous queries. The disadvantage of this approach is that it creates intermediate results and possibly increases delay to the final result compared to stacks fixed to certain sequences. For example, if we have two registered sequence queries, one for A→B→C→D→E and the other for A→B→C we can cache [15] and reuse the result of A→B→C. Therefore, it is important to measure the frequency of sub-sequence patterns [54] in multiple queries and find the optimal n-way joins to exploit sharing without generating useless intermediate results. This is an interesting problem we will pursue as future work. FOLLOWED-BY clause can also be used to express event sequences, but we find this phrase to be limited in expressing long sequences as it needs repetition.

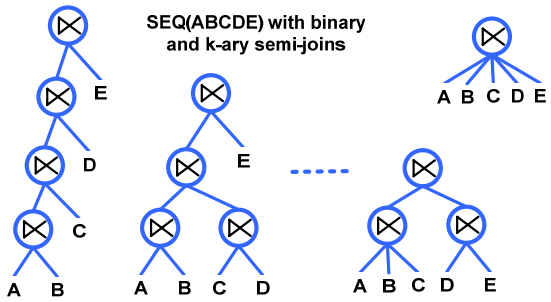


Figure 7: Different options to implement long event sequences (e.g. A, B, C, D, E) using binary and/or multi-way semi-joins.

4.3.2 Query Plan Generation and Optimizations

We adhere to two query plan generation principles called Precision Sharing (PS) proposed in [36], which set the criteria for efficient algebraic plans with multiple queries.

PS1: *"For each tuple processed, any given operation may be applied to it or any copy of it at most once."*

PS2: *"No operator shall produce a tuple whose presence or absence in the dataflow has no effect on the result of any query"*[36.]

PS1 tries to avoid duplicate processing on the same data whether it is the same tuple moving through the plan or a copy of it generated internally. PS2 suggest early detection and removal of tuples that do not contribute to the final results or output to reduce redundancy in processing. An efficient plan should satisfy PS1 and PS2 criteria. We explain how we achieve these goals next.

To demonstrate we pick three sample queries, which we use in our personalized coupon offering (i.e. campaign management) scenarios. The first query Q1 in Figure 8 returns the number of coupons *created* within the last 20 minutes from stream A and the second query Q2 returns the number of coupons *viewed* within the last 20 minutes from stream A. The third query Q3 returns coupons which have been *viewed* within 20 minutes after they are *created*. The plans for query Q1, Q2, Q3 separately are shown from left to right in Figure 8. The plan for Q1 will first filter the coupon "created" events and then send them to *Count*() function and the plan for Q2 will do the same for "viewed" events. Query Q3 plan has one stream copy and two stream select operators. Since selection criteria may not be known at runtime the copy operator simply makes two copies of the input stream and sends them for selection to the following operators. If registered at different times, one can let these three queries run separately or in an isolated way.

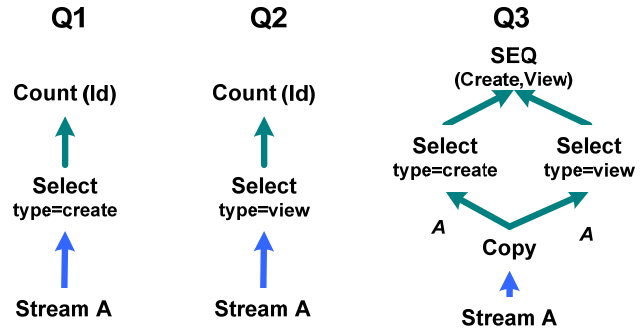


Figure 8: Three isolated query plans corresponding to three registered queries.

A naïve way to generate a single, unified plan would be to make three copies of the input stream and just send each copy along the path of each continuous query. However, the memory and processing overheads of this naïve approach would be drastic, since every tuple in the input stream is copied three times. Our goal is to come up with a new plan which can satisfy PS1 and PS2 and remove redundancies. We can easily observe that while the three queries have common predicates in the naïve plan, SELECT on the predicate "type = created" will be performed on copies of the same tuple in stream A twice, once for Q1 and once for Q3. The same is true for "type = viewed" predicate. Thus, this plan violates PS1 since the same operator will perform on the same data twice.

The plan shown in Figure 9(a) makes a one-step improvement over the isolated plans in Figure 8 to satisfy PS1. This plan removes the duplicate processing on selection operators among multiple queries. The input stream would be copied once and copies would be send to the selects with "type = created" and "type = viewed" predicates, respectively. We assume unbounded windows in this example. Since each tuple is processed at most once by the same operator and predicate, the plan satisfies PS1. However, the plan still has duplicate tuples (create, view, print) flowing through it. Assume that the arrival rate of stream A is λ. The copy operator will send λ tuples to its two downstream operators. If the selectivity of σtype= created is x1 and σtype= viewed is x2, then predicate "type = created" would send x1.λ tuples to its output queues. Also note that tuple 5 (print), which is useless for the output, is still copied from stream A and sent to both selectors. Therefore, this plan violates PS2 as tuples that do not contribute to the result are still being processed.

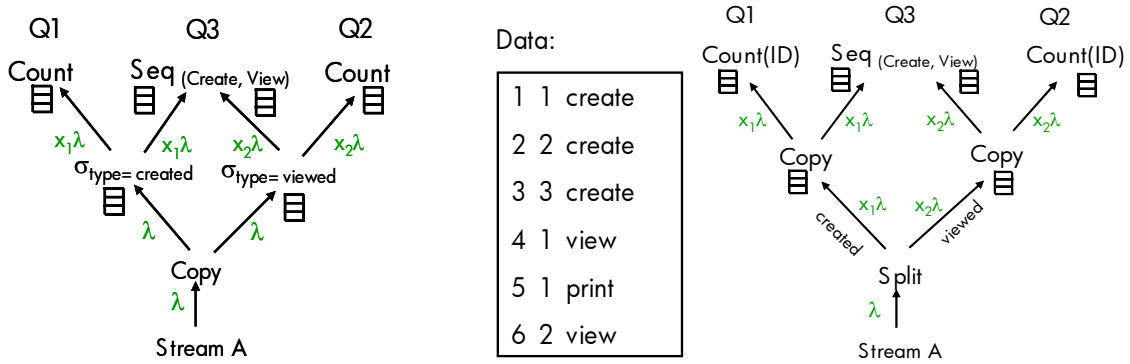


Figure 9: (a) The naïve query plan which satisfies only PS1. (b) Updated plan that satisfies both PS1 and PS2.

To remove the overheads from the previous plan, we improve the query plan for multiple queries as shown in Figure 9(b). This plan uses a SPLIT operator and selection push-down technique [36]. This SPLIT operator has two functions: first is to filter useless data and second is to select & route the filtered events to the related output queues. In this case, *print* event will be filtered out after the split operator. We can visually compare the number of tuples in the flow and see that it is less than

the number of tuples in the previous plan. After split operator, the left side is already $x1\lambda < \lambda$ and the right side is $x2\lambda < \lambda$. Thus, this plan satisfies PS2 since it removes the useless data as early as possible. The plan will remain efficient due to the split even if pushed-down selectors are not offloading the downstream due to low selectivity. Based on the analysis of flow, we can see that although we introduce new operators, the overall computation cost is actually reduced due to increased efficiency and the unbound nature of the streams. In our current implementation, generated sub-streams are directly "pushed" to the attached operators, therefore we do not use a PUBLISH [61] or INSERT [7] clause in our EPL. When queries are nested in a plan, if some queries are expected to "pull" the intermediate results, there would be a need to uniquely name outputs of operators so that they can be identified and referenced in the downstream. Also note that in our baseline DSMS engine CAPE, plans were generated manually for a single query while we support multiple queries here.

The split operator is internally implemented as shown in Figure 10 with "select and route" operators. New tuples are first sent to a selector to insert events that are needed downstream (created, viewed) and to filter the rest. Next, inserted events are routed to the correct downstream operator by the router. A UNION operator may also be used to complement SPLIT operator and re-merge some of the sub-streams [35,2]. This operator could be avoided by careful, gradual splitting of the stream not to violate PS1 and PS2 rules for multiple registered queries. However, use of UNION may be motivated by online query registration scenarios where previously split sub-streams are needed again by a new query in merged form and starting from the original stream is inefficient.

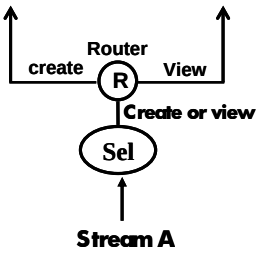


Figure 10: Internal implementation of the Split operator.

We did not describe how to slice streams into time windows and compose by exploiting sharing among similar queries that only differ in their window specifications (e.g. return "created" within 5 and 20 minutes). This issue has been discussed in some of the recent related work [37,56]. We also do not publish an algorithm for managing online, multi-query registrations [59] in this paper as our focus is on business integration of CEP. However, we note that this is a complex and interesting problem where the algorithm requires choreography among shared predicates, overlapping windows, and other QoS constraints set on the systems by users. We believe, only adaptive methods that can re-arrange the query plan under dynamic conditions [37] are appropriate, since a single query joining and leaving the system can greatly affect computations. The goal is to minimize changes to the concurrently used plan, so that scalable online registration is achievable.

4.3.3 A Bottom-Up, Progressive Scheduling Algorithm

This section describes the operator scheduling (i.e. execution ordering) inside our query plan. Since each operator in ReCEPtor may be involved in different queries due to sharing, we needed to reexamine scheduling algorithms in CAPE, namely the Round-robin and the Queue-size based scheduling. Figure 11 compares different scheduling alternatives for the previous query plan in Figure 9b. Round-robin chooses operator execution order based on operator identifiers. For instance, the order can be op1→op2→op3→op4→op5. However, this order can be too random in large scale with respect to the real data flow direction. If the input to an operator is empty such as op1 and op2, they could not generate any output results. Input queue-size based scheduling chooses the operator with the maximum queue first. This can lead to starvation of operators with smaller input queues

although they may be serving queries with strict QoS requirements. We currently focus on guaranteeing data flow and treat queries equally, so that we can continuously output results for all three queries in Figure 11.

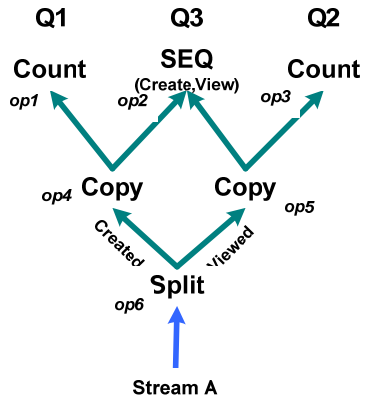


Figure 11: The progressive, bottom-up operator scheduling inside a query plan (from Fig.9b).

To guarantee data flow in the correct order, we propose bottom-up scheduling. This method triggers the operator executions (fetch, process and output input tuples) in a bottom-up tree traversal order. It pushes the input streams in the correct direction towards the outputs and guarantees progress for all operators. For the plan in Figure 11 the scheduling order would be op6→op4→op5→ op1→op2→op3. Queries may be treated unequally for QoS-aware scheduling [2].

5 Proof-Of-Concept Prototype and Results

We developed our CEP engine –ReCEPtor- using Java (with Eclipse IDE) and on top of CAPE stream processing engine. We specified the input stream schema and the system configuration parameters using XML files. To bridge the gap between CEP engine and business applications [25] we implemented the service-based middleware described in Section 3 and implemented proof-of-concept web-based and console-based applications that will be detailed in this section. We start with preliminary performance results and then introduce applications that benefit from CEP engine.

5.1 Performance Results

We ran a preliminary performance analysis of different components in our CEP architecture. First, we scheduled asynchronous web service calls to different tiers in our 3-Tier architecture on a single HP workstation with a 3.20GHz Intel Xeon CPU and 3.5GB RAM. In each round, we included one more tier and reported results in Table 1. We used Microsoft IIS web server, SQL server 2005 database, and Biztalk middleware for messaging and orchestration. We did not tune the worker thread pools and network connection settings, which could have affect on our performance results. We also didn't include applications in these measurements, which would constitute the 4th tier.

# Requests	1-Tier (WS)		2-Tier (WS+DB)		3-Tier (WS+Biztalk+DB)	
	Total RT (ms)	Reqs/sec	Total RT (ms)	Reqs/sec	Total RT(ms)	Reqs/sec
1	484	2	515	2	1000	1
10	500	20	560	18	1800	6
50	530	94	687	73	6000	8
100	625	160	781	128	7200	14
200	700	286	1578	127	12000	17
500	1040	481	2546	196	30900	16
1000	1640	610	3234	309	66000	15

Table 1: Performance results for 3-tiers in CEP architecture.

Results in Table 1 show that while web service tier has about 500ms synchronous call latency, hundreds of asynchronous requests (e.g. 610) can be completed each second. When we add the database tier to the web service, it adds only minimal latency for light loads (<=100 req/sec) and web service like latency for heavy loads (>= 100 reqs/sec). Thus, the throughput for 2-Tier (WS+DB) is about half the 1-Tier (WS-only) around 300 reqs/sec. Yet, the biggest performance hit comes from the 3rd tier, which is the Biztalk middleware for workflow management and messaging. We can only obtain a few tens of (~15) reqs/second when we add the 3rd Tier (WS+Biztalk+DB). For simple workflows like ours we could have omitted the visual orchestration and used only ports in Biztalk. However, as we plan to extend our application logic later, we wanted to maintain this layer and report the potential performance overheads.

We also tested the CEP engine with queries that would push the throughput side. The first query simply does value-based filtering over a stream (**`Select * From Stream A Where A.value > 5`**). We ran this query for 13 seconds and 173 output tuples (i.e. 13.4 reqs/sec) were generated from 655 random inputs (Selectivity=0.26=173/655). While the outputs of this query are not complex events we know that CEP could possibly register events at this rate to the event repository through the 3-Tiers mentioned before, which could support 15 reqs/sec as shown in Table 1. We ran a second aggregation query over the stream (**`Select Count(*) From Stream A Within 6 Every 2`**) for the same duration and got 1381 output tuples from the 2440 input tuples. This maps to 106.8 reqs/sec, which could only be supported by the 2-Tier version of the middleware at this time. We need to tune the application logic implemented in Biztalk server to be able to support this rate of arrival. We conclude that we could process ~15 requests/second through the 3-Tiers of our prototype (web services, orchestration, and database) at this time without tuning.

5.2 Applications

The web console shown in Figure 12 can be used by managers to receive alerts. This web console uses Asynchronous Java and XML (AJAX) to call the orchestration's GetNewEvent method, which returns the newly submitted complex events by the CEP engine. We also used a CORBA bridge to connect CEP engine implemented in Java to the business process shown previously in Figure 4 and implemented in Microsoft.NET and to demonstrate our capabilities for heterogeneous platform integration. The web console was implemented in ASP.NET. It was only tested with one sample event (shown in Fig.12) and not tied to the retail and banking applications described in Figure 13.

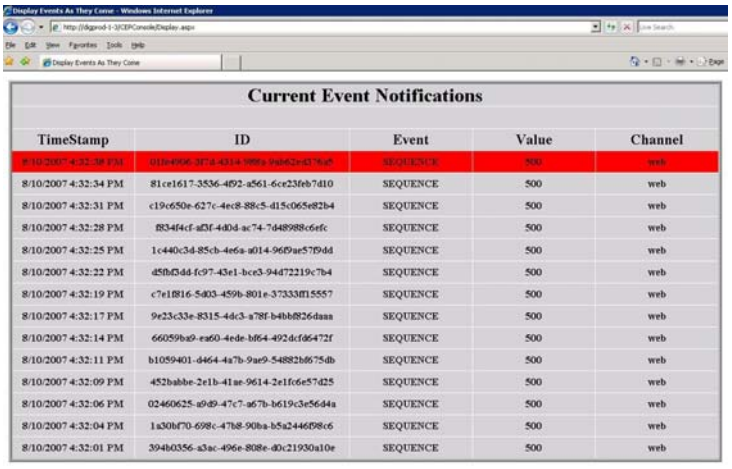


Figure 12: Web-based console to receive alerts on complex events sent by the CEP engine.

A recent analysis [24] finds that multi-channel customers of a retailer (e.g. who use the web, store, phone channels) spend more than one-channel customers and that online buying does not replace in-store buying. Therefore, multi-channel use can be a good indicator of customer loyalty [49]. To demonstrate how our CEP architecture can support multi-channel use cases, we implemented two

similar console-based applications. One demonstrates a high-value customer tracking application for banks and the other demonstrates a personalized, real-time micro-campaign management application for retailers. Both GUIs provide good interaction with CEP engine. In the GUI we want to show multiple query registrations and multiple query outputs. The panels on the right side shows the arriving input data stream. We assume that streams from multiple channels are merged into a single stream, which is logical if one needs to detect event correlations [58]. There are more input data accumulating in the right panel after some time. There is a query entry panel on the bottom left. Here user can register a new query by typing a query as an EPL statement and submitting it to the engine for parsing and plan creation (or integration). The top left panel shows application statistics. Note that these applications could also be implemented as a web portal similar to Figure 12, but we wanted to highlight here the interaction of CEP engine with different application types.

Figure 13(a) shows the proof-of-concept console application to demonstrate a multi-channel banking scenario. On the right-hand-side we see the input stream arriving to the management console through a TCP/IP socket. This application can serve as a front-end for a system that tracks streams of data for customer transactions such as deposits withdrawals as well as promotions offered to customers at different banking channels including the Web, ATM, teller, and call center. On the query panel we can register continuous queries, which get parsed before the application is started. Multiple queries have been registered in this application with the CEP engine to detect complex events and track statistics. We use the same template for the following retail application. The top-left panel shows the window-based statistics on the stream about offers and transactions. The results are refreshed every few seconds. The left screen shows the results of sequence queries, which are displayed as teller alerts. If the customer first views a fixed-rate Certificate of Deposit (CD) offer in the bank's web site (i.e. web channel) and then comes to the store to deposit certain amount of money, then the system alerts the teller. This represents a multi-channel, high-value prospect detection scenario and the teller response would be to suggest this customer to deposit the money to a CD fund rather than his checking account. We hope to integrate and test this use case scenario with HP's Adaptive Bank platform in the future.

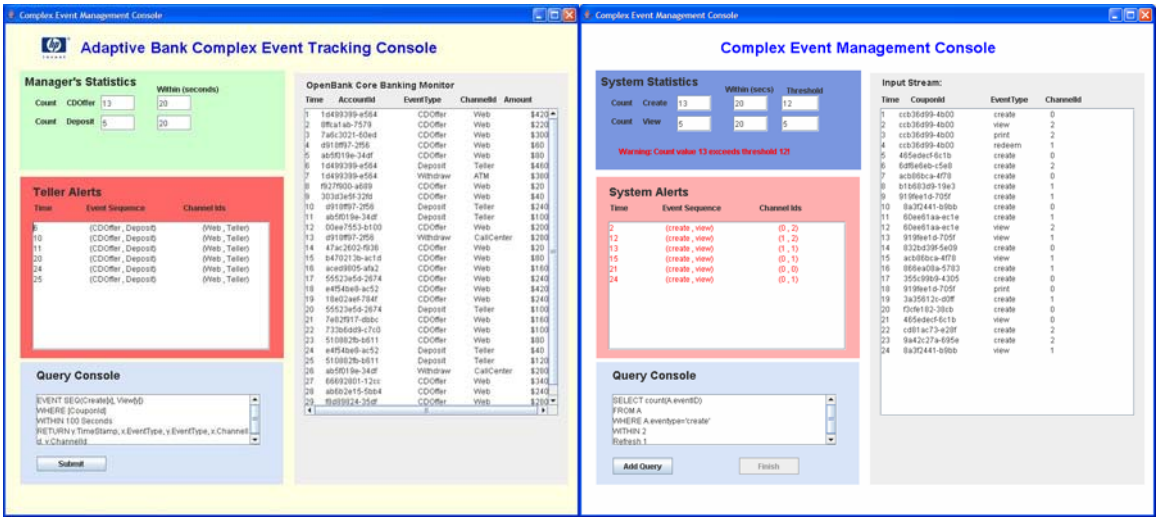


Figure13: Two similar proof-of-concept console applications demonstrating (a) a multi-channel banking scenario and (b) a real-time, retail micro-campaign tracking scenario.

Figure 13(b) shows another console application use case which represents a real-time micro-campaign tracking scenario. On the left top we show results of two continuous queries and the results are refreshed every few seconds. The first query counts the number of the coupons which have been created within 20 seconds. When the value is greater than the set threshold (12), we show a threshold-violation warning in red color. This alert represents a rate control for the offers in order

not to exceed a certain allocated budget. The second query counts the number of the coupons viewed with 20 seconds. Similar to the banking application, coupon application has an alert panel, which tracks coupon event sequences over multiple channels. As the input stream populates the right panel, we start seeing more (create, view) sequences on the left side. We hope to integrate and test this application with HP Laboratories' Retail Store Assistant (RSA) platform [10] in the future. It would also be interesting to merge the query plan trees for these different applications to address issues related to "query forests".

5.3 A Closer Look at Complex Event Categories

We list three complex event categories and give examples for each category to demonstrate use cases that can be supported by ReCEPtor.

- *Threshold-contract violations:* we exemplified this class with retail out-of-stock situations [58]. Another example would be the abnormal difference between the running-average service time and current service time of one instance of a certain business process. Note that the current instance's execution time is compared against a non-static value, which requires use of a stream AVG operator with the join.
- *Sequence patterns in data streams:* we exemplified this class with the credit card fraud case before. Any lifecycle management scenario with a state machine could constitute an example, since a certain sequence of events is expected to occur in the lifecycle.
- *Detecting non-occurrence of events in data streams:* we exemplified this class with the shoplifting use case given in [26] before. Another example would be the non-occurrence of an exit reading in a toll bridge for a car [7], which had an entry reading 20 minutes ago. Either the car is in trouble on the bridge, or the system is not working properly and missing tolls, or there is fraudulent use. Yet another example would be the non-occurrence of a truck arrival at a warehouse long after it leaves the distribution center. In addition, the temperature of goods in those trucks may need to be maintained at a certain level during transport to assure fresh arrival at the destination [61], which would combine the first use case with this one.

Sensors can be used together with CEP systems to facilitate monitoring and control of these "situations" [3] that have business-level impact. These examples merely serve to point to the endless use cases possibilities and do not constitute a comprehensive list as this is not possible.

6 Related Work

Numerous data stream management systems (DSMS) have been developed over the last decade [43,19,46,2,19,28]. STREAM [43] pioneers this field as it describes the semantic foundations of continuous query languages [7] and identifies fundamental challenges such as efficient resource management, operator scheduling, window sliding and approximation techniques in handling data streams. TelegraphCQ [19] focuses on adaptive query processing over high-volume and highly-adaptive dataflows. Eddies route tuples among query operators and control dataflows, which enables TelegraphCQ to do continuous re-optimization of the query plan on a tuple-by-tuple basis. Eddy outputs the tuple when all processing is complete. Aurora [2,1] supports most of the features supported by the previous two systems including novel stream operators, static data binding and continuous query optimizations. However, it differentiates by accepting QoS specifications, prioritizing boxes in data flow graphs and "train" scheduling to meet latency and other QoS goals. None of this early work focused on detecting complex sequences in event streams or business process and SOA integration issues.

CEP also known as event stream processing, which derives its roots from data stream processing, has been the focus of more recent work [17,26,61]. SASE [26] is an NFA-based CEP engine that uses a stack implementation to filter and correlate events. They demonstrate capabilities of their

engine via shoplifting and misplacement scenarios of RFID-tagged items in a retail store. They contribute new event language semantics for sequences and non-occurrence of events. They also describe optimizations for handling large windows and reducing intermediate results. However, they do not discuss plan-level sharing and other issues related to multiple continuous queries in one DSMS. In Cayuga [61], researchers demonstrate the use of their NFA-based CEP for Web feeds and stock prices. Similar to ReCEPtor, Cayuga can run multiple queries concurrently and events sequences would be implemented by NEXT-FOLD constructs. They claim an impressive processing rate of over 1000 events/second for up to 400K active subscriptions [61]. Both SASE+ and Cayuga support Kleene closure. We focused on coupling ReCEPtor with business processes for this paper. Therefore, we can only claim a few tens to hundreds requests/second processing rate, yet end-to-end through the 3 tiers (web services, business process and database) of our architecture. We will work on performance tuning, optimizations and a comprehensive evaluation in the future to achieve larger-scale and higher-speed processing rates. CEDR [17] suggests integrating pub-sub with CEP and DSMS as they share a common processing model and focuses on supporting weak-to-strong consistency guarantees. While this step is a natural progression, we would not agree in merging more middleware components into the CEP engine to maintain separation of roles.

The competition for CEP engines is also beginning to heat up [27] in the industry as new companies enter into the market including Coral8 [20], StreamBase [50], Progress-Apama [6], and many others. http://www.complexevents.com lists major players in this emerging industry.

Detecting “non-occurrence” or absences in event streams is also a challenging problem. Non-occurrence -also called negation [26]-can be cast as a sequence detection problem where the missing event is represented by the occurrence of either a timeout or an out-of-sequence event which violates the expected order. We plan to implement the negation operator and the set of use cases mentioned previously in the future. Out-of-order event arrivals [34] are also common in real deployments. We do not talk about data cleaning, preprocessing and reordering issues in this paper.

Enterprise-scale CEP systems would be used concurrently by many users. This is especially true for service-based architecture and a CEP service designed for SOA. In a CEP service, it may be crucial to impose a beginning and an end to continuous queries registered with the shared system and possible do state maintenance for seamless continuum [10,30]. This concept of “query sessions” or query lifespan management [41] (note: we do not mean tuple lifespan) is a challenging and interesting research problem. Online auctions could constitute a good use case scenario for query session concept [53]. In shared CEP systems, “query templates” [56,35] could be saved and reused among over-lapping and non-overlapping query sessions by one or more applications. StreamMill [16] shares our notion of a query session by defining an EXPIRE clause in the event stream language.

Multi-query optimization (MQO) [32] is an interesting and emerging research area for CEP performance and scalability as well as other traditional database systems. For DSMS, MQO could either be done once at compile-time [2] or incrementally while the system is running [37]. While users insert and delete queries to and from the query plan [59] the system we can measure the frequency of sub-sequence patterns from multiple queries and find the optimal multi-way joins [15,29,55] to exploit sharing without generating useless intermediate results.

Integration of a company’s data sources alone does not help with the bottom-line of a company, since the amount of data is overwhelming and keeps growing [31]. Creating data warehouses and views are good practices, yet they come with similar problems [35]: First, they are after-the-fact, i.e. not fast enough for many real-time applications. Second, cubes, data mining models and new algorithms are not easy to develop and understand as they require special statistical expertise combined with business domain knowledge. However, it will be imperative for CEP engines to communicate with databases, data warehouses and cloud services in many real-life scenarios.

Our work is related to past work in active databases [42] and event-condition action (ECA) rules. While active database research resulted in significant contributions such as triggers and pioneered new research areas including data DSMS, it did not provide the time-constraints provided by many DSMS today. IBM researchers proposed a CEP engine called AMIT [3] and described how to do "situation" management based on ECA techniques.

7 Conclusions and Future Work

It is crucial for many organizations today to process events streams from multiple applications in real-time to quickly respond to critical situations. These organizations are adopting service-based architectures to increase their agility, yet the CEP engines developed have not been following this trend. We described the design and implementation of our CEP architecture that uses SOA principles and we integrated our CEP engine into this architecture. ReCEPtor can sense complex event patterns and collect statistics from raw data streams. Then, it sends complex event messages to other applications and services either directly or through other middleware for further processing. We showed how to use an off-the-shelf BPM system and other engines with ReCEPtor to develop an "enterprise nervous system". We focused more on interoperability, flexibility, efficiency, and reusability aspects in this paper rather than performance and new language semantics.

Our algebraic approach allows us to implement queries for long event sequences while also conforming to precision sharing principles, which eliminate duplicate computations and tuples from a query plan. We used bottom-up scheduling to achieve progressive data flow inside the plan to reduce delay to any particular query. As proof-of-concept we presented two running examples: real-time personalized retail campaign management and high-value financial customer detection over multi-channel. We developed both web-based and console-based applications and integrated them with ReCEPtor. We can already process tens of requests/second through the 3-Tiers of our prototype (web services, orchestration, and database) end-to-end and without tuning and we are working on achieving higher speeds.

There are many improvements and research problems we would like to address in the future. For example, in our query execution we treat each query equally. However, different queries have different processing cost and therefore tuple consumption rates. Dynamically adapting the load on different operators via scheduling [13] to assure smooth dataflow over the query plan is an interesting research problem. Based on the different QoS goals such as latency, throughput or jitter, we can collect statistics about every operator at execution time and adapt scheduling to meet these goals. QStream [47] uses a real-time capable operating system for reservations and controller-based adaptation to assure steady-state output from DSMS.

CAPE supports plan distribution among multiple machines for scalability and load balancing [38]. We plan to utilize this feature to support distributed operation for ReCEPtor to address high-speed event detection and processing in large-scale enterprises. Intermediate results may have to be shared among the sub-plans distributed over different nodes. We will evaluate architectures for flexible distributed query processing [35] and other multi-level caching techniques [9,11] to make good design choices. Other high-load handling techniques include load shedding [39,14], query admission control [47], disk buffering [38], and compression using binary formats [22]. Data mining (linear regression, decision trees, clustering) [14,54,23] and change detection [18] over event streams are also interesting research directions.

Acknowledgements

We would like to thank Menaka Indrani, Kiran Kadambi, Krishna Sheelam for helping with the project development. We also thank Lars Rossen, Pankaj Mehra, and Prof. Elke Rundensteiner for useful discussions and suggestions.

8 References

[1] D. 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, 12(2):120–139, August 2003.

[2] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, J. Hwang, A Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik. Aurora: A data stream management system, In ACM SIGMOD 2003, New York, USA, Page 666.

[3] A. Adi and O. Etzion. Amit-The situation manager. VLDB Journal(13), 177-203, 2004.

[4] M. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra, Matching events in a content-based subscription system, PODC 1999, 53-61.

[5] Albek, E. Bax, E. Billock, G. Chandy, K.M. Swett, I., An Event Processing Language (EPL) for Building Sense and Respond Applications, In Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS), April 2005, Page136.

[6] Progress-Apama, http://www.progress.com/apama

[7] A Arasu, S Babu, J Widom, The CQL continuous query language: semantic foundations and query execution, VLDB Journal 15(2): 121-142, 2006

[8] I. Ari, E. L. Miller, Caching support for push-pull data dissemination using data snooping routers, 10th IEEE International Conference on Parallel and Distributed Systems (ICPADS) 2004.

[9] I. Ari, A. Amer, R. Gramacy, E. L. Miller, S. Brandt and D. D. E. Long, Adaptive Caching using Multiple Experts, Informatics, volume 14, pages 143-158. Carleton Scientific, 2002

[10] I. Ari, J. Li, R. Ghosh, M. Dekhil, Providing Session Management As a Core Business Service, In WWW 2007.

[11] I. Ari, M. Gottwals, D. Henze, Performance Boosting and Workload Isolation in Storage Area Networks with SANCache, IEEE/NASA Conference on Mass Storage Systems and Technologies (MSST) 2006.

[12] K. Akkaya, I. Ari, In-network Data Aggregation in Wireless Sensor Networks, The Handbook of Computer Networks, Volume 2, John-Wiley & Sons

[13] B. Babcock, S. Babu, R. Motwani, and M. Datar. Chain: operator scheduling for memory minimization in data stream systems. In ACM SIGMOD, pages 253–264, 2003.

[14] B. Babcock, S. Babu, M. Datar, R. Motwani, Widom J., Models and issues in data stream systems, ACM PODS, 2002, June, pp 1-16.

[15] S. Babu, K. Munagala, J. Widom, R. Motwani, Adaptive Caching for Continuous Queries, In Proceedings of the 21st International Conference on Data Engineering (ICDE) 2005, pp 118-129

[16] Y. Bai, H. Thakkar, H. Wang, C. Luo, C. Zaniolo, A data stream language and system designed for power and extensibility, In Proceedings of the 15th ACM International Conference on Information and Knowledge Management, Virginia, USA 2006, pp 337 - 346

[17] R. S. Barga, J. GOldstein, M. H. Ali, M. Hong, Consistent Streaming Through Time: A Vision for Event Stream Processing. CIDR 2007, pp. 363-374

[18] D. Kifer, S. Ben-David, J. Gehrke, Detecting Change in Data Streams. VLDB 2004, 180-191

[19] S Chandrasekaran, O. Cooper, A Deshpande, MJ Franklin, TelegraphCQ: Continuous dataflow processing for an uncertain world, CIDR 2003.

[20] Coral8, Complex Event Processing: Ten Design Patterns. http://www.coral8.com

[21] S Doraiswamy, M Altinel, L Shrinivas, S Palmer, F. Parr, B. Reinwald, C. Mohan, Reweaving the tapestry: Integrating database and messaging systems in the wake of new middleware technologies, Data Management, LNCS, pp 91-109, 2005, Springer-Verlag.

[22] G. Guardalben, Compressing and Filtering XML Streams, W3C Workshop, 2003

[23] L. Golab, M. T. Ozsu, Issues in data stream management, SIGMOD Record Vol 32.No 2, June 2003

[24] J. Grau, Multi-channel shopping: The rise of the retail chains, E-Marketer Report, March 2006.

[25] T. Greiner, W. Duster, F. Pouatcha, R. Ammin, H. Brandt.D. Guschakowski BAM of Norisbank: Taking the example of application easyCredit and the future adoption of CEP, Proceedings of the 4th international Symposium on Principles and Practice of Programming in Java, pp. 237-242.

[26] D. Gyllstrom, E. Wu, H-J Chae, Y. Diao, P. Stahlberg, G. Anderson, SASE: Complex Event Processing Over Streams, CIDR 2007.

[27] A. Handy, CEP made simple, Coral8 and StreamBase, SDTimes.com, July, 1, 2007, page 30

[28] M. A Hammad, MF Mokbel, et al. Nile: A query processing engine for data streams, ICDE 2004.

[29] M. Hong, A. J. Demers, J. E. Gehrke, C. Koch, M. Riedewald, W. M. White, Massively Multi-Query Join Processing in Publish/Subscribe Systems, SIGMOD 2007, 761-772.

[30] R. Baeza-Yates, C. Hurtado, M. Mendoza., Mining Query Sessions in Search Engines or Query recommendation using query logs in search engines. In International Workshop on Clustering Information over the Web (ClustWeb) March 2004.

[31] B. Inmon, C. Imhoff, G. Battas, The Operational Data Store, John Wiley & Sons 1999.

[32] C. Jin and J. Carbonell, Argus: Efficient and scalable continuous query optimization for large-volume data streams, IDEAS 2006.

[33] J. Julio, A. Banti, F. Banti, A. Brogi, An Event-Condition-Action Logic Programming Language, In Proceedings of 10th European Conference on Logics in Artificial Intelligence (LNAI), Liverpool, UK, September 2006, Vol. 4160, Pp. 29-42.

[34] M. Li, M. Liu, L. Ding, E. A. Rundensteiner, M. Mani, Event stream processing with out-of-order data arrival, ICDCSW 2007.

[35] A. Kemper and C. Wiesner, Hyperqueries: Dynamic Distributed Query Processing on the Internet, VLDB Journal, Pages 551-560, 2001.

[36] S. Krishnamurthy, M. J. Franklin, J. M. Hellerstein, and G. Jacobson. The case for precision sharing. VLDB, 2004.

[37] S. Krishnamurthy, C. Wu, M. J. Franklin, On-the-fly sharing for streamed aggregation, SIGMOD 2006, June, Chicago, USA pp 623-634.

[38] B. Liu, Y. Zhu, M. Jbantova, B. Momberger, E. A. Rundensteiner, A dynamically adaptive distributed system for processing complex continuous queries, VLDB 2005.

[39] G. Luo, C. Tang, P. S. Yu, Resource-adaptive real-time new event detection, SIGMOD 2007, 497-508

[40] P. Maheshwari, S. Tam, Event-based exception handling in supply chain management using web services, IEEE AICT/ICIW 2006.

[41] Markowetz A., Yang Y., Papadias D., Keyword search on relational data streams, SIGMOD 2007.

[42] D McCarthy, U Dayal, The architecture of an active database system, SIGMOD 1989.

[43] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, R. Varma, Query Processing, Resource Management, and Approximation in a Data Stream Management System, In Proceedings of CIDR 2003.

[44] Niblett P. and Graham S., Events and service-oriented architectures: The OASIS web services notification specifications, IBM Systems Journal, Vol 44, No 4, 2005

[45] S. Rizvi, Events on the edge, SIGMOD 2005.

[46] E A. Rundensteiner, et al. CAPE: Continuous Query Engine with Heterogeneous-Grained Adaptivity. VLDB 2004

[47] S. Schmidt, T. Legler, S. Schlar, W. Lehner, Robust real-time query processing with QStream, VLDB 2005, pages 1299-1031.

[48] W. M. Schutz, Getting Started with Complex Event Processing Nodes, CEP Primer v1.2, IBM

[49] M. Stone, M. Hobbs, M. Khaleeli, Multi-channel customer management: the benefits and challenges, Journal of Database Marketing, Vol 10, 1, 39-52.

[50] M. Stonebraker, U. Cetintemel, S. Zdonik, StreamBase Systems Whitepaper. The eight rules of real-time stream processing, http://www.streambase.com

[51] H. Su, J, Jian, E. A. Rundensteiner. Raindrop: A uniform and layered algebraic framework for XQueries on XML Streams, CIKM 2003.

[52] StreamSQL, http://www.streamsql.org

[53] PA Tucker, K Tufte, V Papadimos, D Maier, NexMark–A benchmark for queries over data streams

[54] Unal A., Saygin Y., Ulusoy O., Processing count queries over event streams at multiple time granularities, Information Sciences, 2005, Elsevier.

[55] Stratis Viglas, Jeffrey F. Naughton, Josef Burger: Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources. VLDB 2003: 285-296

[56] S. Wang, E. Rundensteiner, S. Ganguly, S. Bhatnagar, State-slice: new paradigm of multi-query optimization of window-based stream queries, VLDB 2006, pages 619-630.

[57] Mingzhu Wei, Ming Li, Elke A. Rundensteiner, Murali Mani: Processing Recursive XQuery over XML Streams: The Raindrop Approach. ICDE Workshops 2006: 85

[58] E. Wu, Y. Diao, S. Rizvi. High-performance complex event processing over streams, SIGMOD 2006, pp. 407-418

[59] KL Wu, SK Chen, PS Yu, Interval query indexing for efficient stream processing, CIDR 2004

[60] X. Xiong, H. G. Elmongui, X.Chai, W. G. Aref, PLACE: A distributed spatio-temporal data stream management system for moving objects, VLDB 2004.

[61] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, W. M. White: Cayuga: A General Purpose Event Monitoring System. CIDR 2007: 412-422