Integration of Cognitive Technologies into Complex Event Processing to Enable Large Scale and Dynamic IoT Applications

Approved: _____ Date: _____
Committee Chair

Approved: _____ Date: _____
Committee Chair

Approved: _____ Date: _____
Committee Chair

Integration of Cognitive Technologies into Complex Event Processing to Enable Large Scale and Dynamic IoT Applications

A Thesis
Presented to
The Graduate Faculty
University of Wisconsin-Platteville

In Partial Fulfillment
Of the Requirement for the Degree
Master of Science
Computer Science

By
Gregory J. Saelens
2013

ACKNOWLEDGEMENTS

I want to thank everyone who has helped me in the completion of this thesis and who helped me get this far.

Special thanks go to my girlfriend, Kathrin Mierenfeld; my mother, Elvira Saelens; my sister, Sandra Saelens; Christine Mierenfeld; Adrian Ilgen; and Liz Einsig Wise, who all proofread this thesis.

I want to thank my advisor and supervisor, Dr. Walter Waterfeld, for the support he gave me before and during this thesis.

I dedicate this thesis to my niece and godchild, Amberlee Brianne Pick. Always pursue your happiness through hard work and studies, and never give up.

"It always seems impossible until it is done"

— Nelson Mandela

ABSTRACT

Integration of Cognitive Technologies into Complex Event Processing to Enable Large Scale and Dynamic IoT Applications

Gregory J. Saelens

Under the Supervision of
Prof. Dr. Ralf Hahn
Prof. Dr. Bernhard Humm
Prof. Dr. Joseph Clifton

Statement of the Problem

In recent years, the IoT has grown from a skeletal concept to an increasingly vital reality. This growth can be attributed to the huge increase in the number, variety and extent of mobile and sensor technology rapidly permeating almost every aspect of modern life. Communication among and between various devices and servers is the key element in the IoT.

An Event Driven Architecture (EDA) provides the preferred environment for the communication requirements of the various sensors and devices that make up the IoT due to their real-time transmission and analysis capability. However, small systems can become unmanageable due to the growing number of events recorded by the ever-increasing number of devices. Because these devices work with many different standards, integrating them into a single system causes additional complexity. Statistical analysis alone, because of its nature, may not reveal all the layers of information hidden in the data.

Methods and Procedures

In the large European research project iCore, the use of cognitive technologies is pursued to manage the heterogeneity of devices and to provide appropriate views for different users and applications. In order to make those usually computing-intensive algorithms scalable, the use of Complex Event Processing technology (CEP) is proposed.

This master's thesis investigates different mechanisms for the integration of cognitive algorithms into CEP. This includes the utilization of CEP to preprocess low level events from a large number of sensor events, consolidating them into a smaller number of high level events which are capable of being processed by cognitive algorithms. It also includes the extension of event processing languages with the appropriate functionality needed in cognitive algorithms, thereby allowing all or a portion of the cognitive algorithm to be executed in near real-time directly on the event streams received from the IoT sensors.

A general concept for integrating cognitive algorithms (machine learning algorithms that process information based on a knowledge, or memory, built from existing data) into CEP is developed based on these investigations. For selected cognitive algorithms, the appropriate integration with CEP will be realized. Their scalability is then evaluated against a selected IoT use case and compared to traditional cognitive algorithms.

Summary of Results

The event processing environment contains a wide variety of features, each created to meet different demands and possessing various abilities, that can be utilized for the realization of cognitive technologies.

Individual analyses of these available features led to the generation of a methodology for realizing cognitive algorithms in CEP. By then realizing the implementation of two algorithms from the field of IoT, and using the features available in CEP, the general capabilities of this combination of technologies are demonstrated.

This combination of IoT, cognitive technology, and CEP is a very broad topic that leaves many possibilities for further research, both generally and in detail.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

LIST OF SYMBOLS

API	Application Programming Interface
BAM	Business Activity Monitoring
BNF	Backus-Naur Form
CEP	Complex Event Processing
CRUD	Create Read Update Delete
CPU	Central Processing Unit
CVO	Composite Virtual Object
DBMS	Database Management System
DCL	Data Control Language
DDL	Data Definition Language
DML	Data Manipulation Language
DSMS	Data Stream Management System
EDA	Event Driven Architecture
EQL	Event Query Language
GUI	Graphical User Interface
HDFS	Hadoop Distributed File System
iCore	Internet Connected Objects for Reconfigurable Ecosystems
IS	Integration Server
ICT	Information and Communication Technology
IoT	Internet of Things
JDBC	Java Database Connectivity
JMS	Java Message Service
KPI	Key Performance Indicator
MOM	Message Oriented Middleware
RFID	Radio-Frequency Identification
SQL	Standard Query Language
UDA	User-defined Aggregate Function
UDF	User-defined Function
UDO	User-defined Operator
VO	Virtual Object
XML	eXtensible Markup Language
XSD	XML Schema Definition

1 INTRODUCTION AND STATE OF THE ART

The number of mobile internet devices, e.g., smartphones, and tablets, is constantly growing. Together with their expanded use, concepts such as the smart home and online shopping gain more importance. This increases the number of objects that access the internet or are tracked online. A smartphone or tablet user may use their device to navigate to their nearest sports retail store; a refrigerator may report its contents and do the ordering if the milk is almost empty; or a customer may track the progress of his or her shoes since they have been ordered online. In the near future, it is expected that cars will be equipped with automatic systems that detect an emergency and send detailed information about the occurrence, together with the car's current location, to rescue services. In an infographic from Cisco Systems Inc., the authors describe the case in which a Dutch farmer has connected all of his cattle using wireless sensors so that he will be informed as soon as one gets sick or pregnant. The same authors have predicted that 50 billion devices will be connected to the internet by 2020 [MacManus, 2011].

These examples show how both the availability and the use of mobile technologies have increased in everyday life, helping to make the vision of the *Internet of Things* (*IoT*) become a reality. The growth in this area is an opportunity for both hardware and software research, development and distribution. Events, such as messages indicating that the amount of milk is low, or that package x has passed sensor y at z, are the preferred communication style for these *IoT* sensors and devices. This is because events and the architecture behind them have the required real-time capabilities that start the coupled mechanisms in real-time. The diversity of the devices and use cases, and the resulting variety in event type and quantity, makes them an interesting and complex field of study.

The large European research project *iCore* pursues the use of cognitive technologies to cope with this heterogeneity of devices and to provide appropriate views for different users and applications. Machine learning algorithms, which facilitate the cognitive technologies, are usually very computing-intensive and require a lot of resources. In order to make the use of these algorithms scalable, the usage of *Complex Event Processing* technology (*CEP*) is proposed in this thesis.

In this thesis the different mechanisms provided by *Software AG's CEP* implementation will be investigated as to how they can be used to integrate cognitive technologies into *CEP*.

Cognitive algorithms are machine learning algorithms that process information based on a knowledge, or memory, built from existing data. This will include the utilization of *CEP* to preprocess low level events from a large number of sensor events, consolidating them into a smaller number of higher level events, which then can be processed by cognitive algorithms. It will also include the extension of event processing languages with the appropriate functionality needed in cognitive algorithms. Through this method, complete or partial cognitive algorithms can be executed in near real-time directly on the event streams from the *IoT* sensors. Based on these investigations, a general model for integrating cognitive algorithms into *CEP* will be developed.

For two selected cognitive algorithms, the appropriate integration with *CEP* will be realized. The scalability of these realizations will then be evaluated against a selected *IoT* use case and compared to traditional cognitive algorithms.

1.1 Statement of the Problem

The amount of data that is used and produced is growing rapidly. More and more information is sent over the World Wide Web, including data from rapidly proliferating small sensors and mobile devices. The *Internet of Things* (*IoT*) is becoming more concrete, and the increased availability of mobile technology has boosted its use in many fields. Smartphones and tablets have brought this new type of mobile technologies into many private homes. Use of *RFID* sensors began earlier in fields like logistics and retail, e.g., package tracking or theft prevention. Communication between the mobile devices and centralized servers happens through the transmission of events, which are uniquely capable of being sent and received in a real-time environment.

The expanded use of such mobile devices inevitably has increased the volume of data produced by them. Besides the directly communicated information inside a single event, e.g., package x has arrived at location y at time z, the events also carry "hidden" information on a higher level, e.g., packages currently arrive at location y with an average delay of z minutes. To use the delayed package as an example, it is statistically easy to calculate the delay, and even the cumulative delay. However, in current real-life situations, the movement of the package is only statically defined, with events from various sensors aggregating only to track its presence – not its accumulating delay. Because of the number of incoming events, even a relatively small system may encounter difficulties in coping with this calculation. Additionally, there is more and deeper information to be revealed that can be used to predict future situations, e.g., whenever the accumulated event a has happened, the packages start arriving at location y with a delay of z minutes three hours later.

1.2 Purpose of the Study

The concept of *Complex Event Processing* is a relatively new field, but it has so far shown ideal capabilities for handling large amount of data. To make the data usable for a human manager, or to automate certain mechanisms, statistical analysis of the data alone might not be sufficient. Information "hidden" inside may benefit from the application of some cognitive technology to reveal previously unseen depths of information or to increase precision in statements derived from the data. The purpose of this study is to discuss the means *CEP* can provide to integrate cognitive technologies that can help analyze large amounts of sensor data. This will be demonstrated with current, practically related algorithms. Combining these technologies to maximize their gain is an interesting, forward-looking and, thus far, barely-covered field.

1.3 Delimitation of the Study

The area this thesis covers is very wide and therefore delimitations are set to narrow the research. These delimitations are:

- This thesis does not involve the design or development of a new cognitive algorithm.
- Implementation of the concept will only be done in Java, *XML* and *Software AG's EQL*. Examples will also use *SQL* to compare it to *EQL*.
- The implementation details of the *CEP* will not be altered nor explicitly discussed. The focus lies solely on the use of the technology.

1.4 Thesis Outline

The topics used in this thesis are complex, and need thorough research up front. The general concepts of *Complex Event Processing* (*CEP*) were originally defined by David C. Luckham in 1996 [Etzion and Niblett, 2010]. Etzion and Niblett provided a revised view on the topic in their book [Etzion and Niblett, 2010] in 2010. They also included current scenarios as examples and a table of current *CEP* implementations in various categories. The *CEP* implementation of Software AG shall be used for demonstration; therefore, it must be analyzed regarding how it implements the defined *CEP* concepts. The *Internet of Things* (*IoT*) is a similarly new concept, first described by Kevin Ashton in 1999 [Ashton, 2009]. According to Mattern and Flörkemeier [Mattern and Flörkemeier, 2010], the *IoT* has remained conceptual since that time, and so the implementations differ in their interpretation of the concept. The basis for this thesis shall be the definition of

the *IoT* and its usage scenarios described in *Internet Connected Objects for Reconfigurable Ecosystems* (*iCore*) project [iCore Project, 2011a].

Regarding the third concept, the cognitive technologies: standard algorithms shall be used. Additionally, at least two algorithms of cognitive technology shall be used that are applicable in one or more of the described *IoT* scenarios.

The usage of cognitive technologies in the *IoT* is not new. However, in order to analyze the advantages of the integration of *CEP* into this environment, current solutions without *CEP* shall be examined.

To conclude the research, the conceptual possibility of using the standard cognitive algorithms inside *CEP* shall be analyzed. Based on the results, a *CEP* implementation of the two selected real-life *IoT* algorithms will demonstrate the concept, and the practicability of this thesis.

2 BACKGROUND AND RELATED WORK

2.1 Complex Event Processing

Complex Event Processing (*CEP*) is a fairly new way of analyzing data and data streams; in fact, one of the first references was made by David Luckham [Luckham, 2002] as recently as 2002. A *CEP* system handles streams of events and creates complex events based on either rules or queries. Using *CEP*, a program is capable of analyzing large numbers of events and reacting not only to an individual event, as traditional systems using events for communication would do, but also aggregating events and analyzing patterns within them.

Many existing computer languages and systems are capable of event processing; however, they use individual systematic approaches, i.e., in most cases, one event is sent to one recipient. This recipient resides on the same host or even inside the same application. It is not easy to place different events into relation to each other without producing specialized code with extensive programming overhead for each instance. Using a single centralized event bus (see Figure 2.1 on page 6) for communicating all events of a certain group allows another producer or another consumer of events to connect with the system. It also provides an easy way to include a *CEP* engine which consumes one type of events, does calculations on them using a predefined syntax, and produces complex events which it puts back onto the event bus.

While Luckham's work concentrates on the rule based processing of events [Luckham, 2002], Etzion and Niblett focus more on query based *CEP* [Etzion and Niblett, 2010], which is also the way *Software AG's CEP* is utilized. In the case of *CEP* engines using query based languages, the engine has some features that are comparable to relational DBMSs, but also has some very important differences. Whereas data is persistent and queries are transient in a *DBMS*, it is the opposite in a *CEP* system. Additionally in a CEP system, time is not perceived as available as an optional or editable data type, but as an essential and immutable element describing the occurrence of the event.

The benefits of *Complex Event Processing* are [Etzion and Niblett, 2010]:

- Implementation is ideal in an environment that produces data through events by sensors.

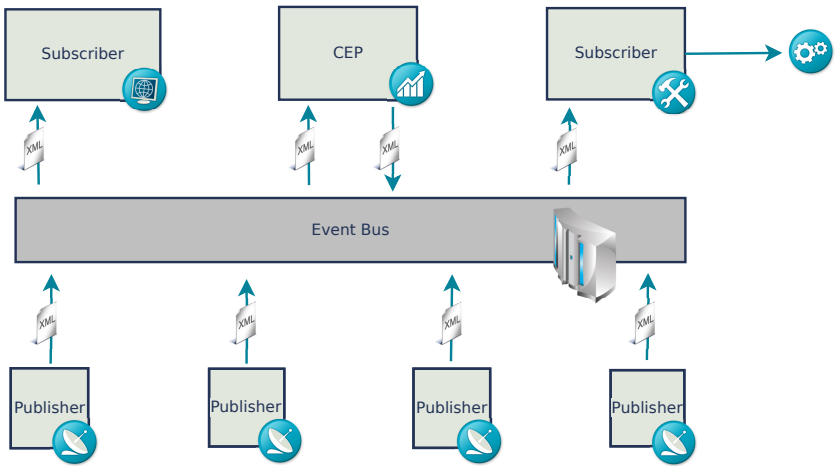

Figure 2.1: Example of an event bus with several publishers, subscribers and a *CEP*.

- Software design is facilitated through an existing event processing engine.
- To a high degree, event producer and consumer are decoupled.
- Systems are able to handle a vast quantity of events ($\approx$ 1 million events per second).
- Data is continuously analyzed, allowing for real-time reaction as compared to batch processes.
- *CEP* allows for scalability and fault-tolerance.
- *CEP* can analyze complex patterns within events to make predictions or analyze trends.

2.1.1 Events

As in the real world, events have become an integral part of computing systems. The Oxford English Dictionary defines an event as "something that happens or is thought of having happened". Inside a computer program, the event is represented as a programming entity that contains the information about a real world occurrence.

Events in computing systems can be simple, such as an implementation of the observer pattern in an object-oriented programming language, as described by Gamma et al. [Gamma et al., 1993]. Languages like Java and C# utilize this pattern in their graphical libraries to communicate events. In this case, the events, along with their producers and consumers – in most cases at least the producer – must be known up front.

A system based on these concepts can handle only a small number of events, and is built simply to react to incoming events rather than analyzing patterns within.1 Events

can be classified as deterministic, when they are an exact description of a real world event, or as approximate, when they are approximations of the real world [Etzion and Niblett, 2010]. Luckham calls the real world events that are represented by events in the system "activities".

According to Luckham, an event has three aspects, which he named Form, Significance, and Relativity:

Form
: This is the structure of the object, which represents the event. It can be a string, or a tuple of data components. A single data component inside an event can be referred to as an attribute. The way in which the form is described is implementation dependent.

Significance
: An event signifies an activity inside the system, which is why Luckham called it the event's significance. That is the specific data for a certain event giving the structure of the event that is defined in its form. The way the significance is stored is implementation dependent.

Relativity
: Every event is related to other events by time, causality, aggregation, or a combination thereof.

An example event given by Luckham is the following event, with its structure in a Java class, showing both the form and the significance:

```
Class InputEvent
{
    Name        NewOrder;
    Event_Id    E_Id;
    Customer    Id;
    OrderNo     O_Id;
    Order       (CD X, Book ...);
    Time        T;
    Causality   (Id1, Id2, ...);
}
```

It shows an event of the type "InputEvent". The identification of the event is given by its *Event_Id* and a timestamp in the field *T*. In some cases the time can also have a span or an end time to represent a time interval. Here, the event itself even provides a way to trace its causality by listing its causes. The other fields are attributes of the event.

In order to use the events, they first need to be created and then inserted into the *CEP*. The creation involves two steps:

Observation
The real life events – activities – are observed without intervention in the flow of events.

Adaption
The observed events are transformed into event objects in the form and layout understood by or configured in the *CEP*.

Events can be created by three different layers within the systems:

IT layer
In the IT layer, any signal that comes from components outside the system can be transformed into an event. Events can also be sent using a *Message Oriented Middleware* (*MOM*) [Curry, 2005] or read from databases. The external system can also be a component running on the same system that communicates using predefined methods.

Instrumentation
This level of event generation is located on the system running the *CEP*. Any activity within that system can be turned into an event, e.g., heartbeats or operating system status messages.

CEP
By processing other events, the *CEP* can create and re-insert its own events. Those can be seen as inferences drawn from events observed from any layer.

Figure 2.2 on page 9 shows the flow of the events through the entire system and the *CEP*. The events can be created in the *CEP*, on the system or in the IT layer, but need to be adapted for the *CEP* to be able to process them. Events from the *CEP* also need the reverse adaption for the system to use them.

2.1.2 Event Relationships

The most important part of an event is its timestamp: it is immutably integrated into it and must be generated by the system that also generates the event. If the timestamp were to be added by the event processing system, its time of arrival at the system would gain more importance than the time of its occurrence. In that case, even minimal differences would significantly alter the processing results. *CEP* differentiates between application time – the time on the clock of the producer of the event – and the system time – the time of the *CEP* engine at the time of arrival of the event. The time reflects Luckham's concept of relativity (shown in section 2.1.1) among the events. It enables the events to be ordered chronologically in relationship to each other.

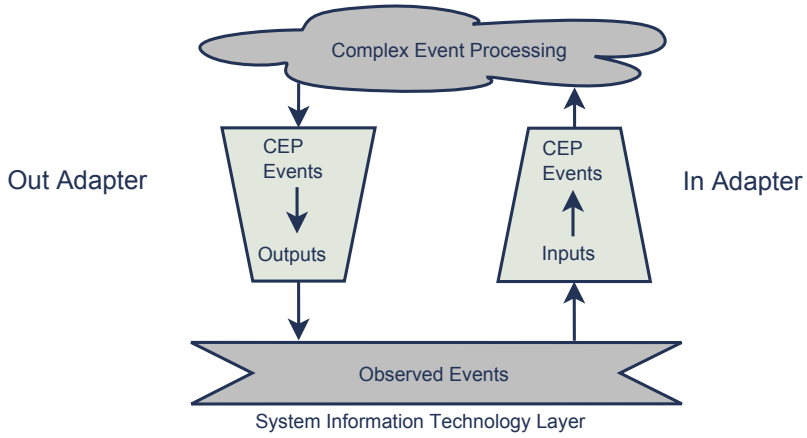

Figure 2.2: Observation of events from an IT layer [Luckham, 2002].

Causality describes the relationship between events whose activities initiated each other. Luckham underscores that this is a computational causality that depends on the target system. This is not comparable to *philosophical or statistical notions of cause and effect*. A is said to have caused B if the activity described and represented in A was the reason for the activity to occur which in turn is described and represented in event B. It is important to understand causality within the events inside a system.

Aggregation is an abstraction level of events in which many events get aggregated into a single one. Usually a causality exists in which an event of a higher abstraction level is created when a set of lower level events have happened. An aggregated event is a complex event: it not only has a single timestamp, but also a time span from the earliest to the latest of its members.

All the relationships are transitive and asymmetric in a mathematical sense. They are also considered a partial ordering rather than a total ordering. That is because none of the three relationships has to determine a specific order between any two events.

An important rule that does state the consistency of causality and time in most systems in the **Cause-time axiom**:
"If event A caused event B in system S, then no clock in S gives B an earlier timestamp than it gives A" [Luckham, 2002].
As can be seen in Figure 2.2, events enter the *CEP* through adapters and may or may not arrive in their causal order. If they do arrive in their causal order the *observation is orderly*.
Orderly observation: *for any pair of events A and B, if* $A \longrightarrow B$ *then ArrivalTime(A)* $\leq$ *ArrivalTime(B)* [Luckham, 2002].

2.1.3 Complex Events

Processing of events happens when common CRUD operations are performed on the events. Creating events, either through aggregation or through analysis of aggregated events, produces complex events [Etzion and Niblett, 2010]. The events aggregated within a complex event – which may have originated from unconnected systems and occurred at different times – are called its members. According to Luckham, events are aggregated into complex events through the use of aggregation rules that recognize significant high-level events in the mass of low-level events. His approach is a purely rule based aggregation of events.

Etzion and Niblett have collected various complex event languages and products in use, and categorized them into three language styles: stream-oriented, rule-oriented, and imperative. *Software AG's CEP* is still listed as *RTM Realtime Monitoring*'s *RTM Analyzer* in this table. Etzion and Niblett's book uses Apama's MonitorScript Event Programming Language, ESPER and StreamBase Stream*SQL* EventFlow, to demonstrate the examples. MonitorScript is an imperative language with syntax similar to that of Java or C++. ESPER and Stream*SQL* EventFlow are both extensions of *SQL*. The language used by Luckham – RAPIDE – is not shown in the table.

An example off the aggregation of single events into a higher level, complex event is shown in Figure 2.3 on page 11. The figure describes an online auctioning house, but it could be applied to other situations as well. It shows five low-level bid events entering the system within a time span of one hour. Each bid event declares that a higher bid was made for a certain item, but more information can also be extracted. The example shows that the events within one hour are collected and aggregated. From the aggregated events, a new event is created with the average, minimum, and maximum bid information from the members. The new complex event does not have a timestamp, but a time span that starts with the beginning of the first member and ends with the end timestamp of the last member. The figure could also be interpreted to show an aggregate of five consecutive low-level bid events. In this example the result would be the same.

Complex Event Processing allows the hierarchical abstraction of events. Through this process, they can become more comprehensible for humans: already, these hierarchies are found in society, business, military, and inside computer systems.

2.1.4 Software AG's CEP

This introduction to *Software AG's CEP* is based on the *RTM Analyzer* tutorial [RTM Realtime Monitoring GmbH, 2010]. In 2010, *Software AG* acquired *RTM Real-Time Monitoring*, introduced *RTM*'s *CEP* into their palette, and subsequently has continued development on it. This *Complex Event Processing* engine is query based. Queries are defined in the *Event Query Language* (*EQL*), a language similar to *SQL* but with an additional language for the description of patterns. The pattern language is different than Luckham's.

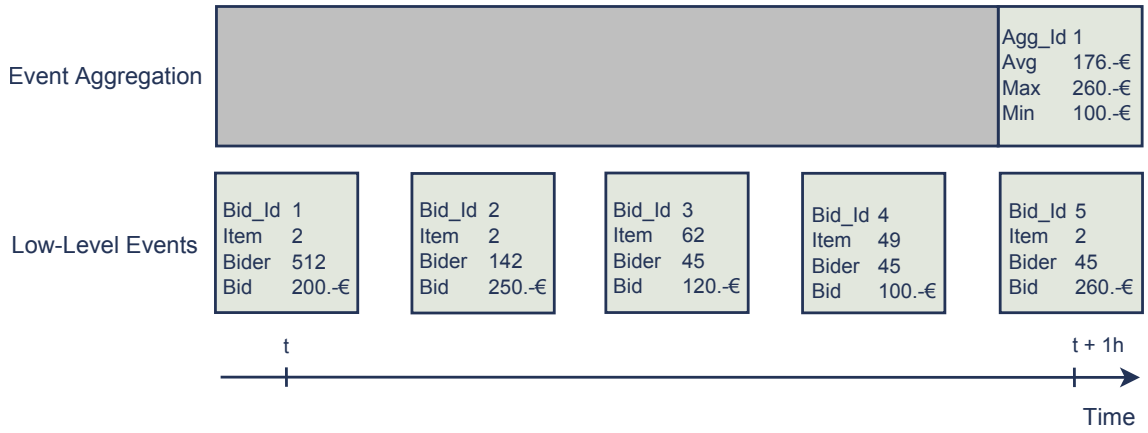

Figure 2.3: Example of an aggregation of events.

The full syntax for *EQL* can be found in section 5.4 in the appendix on page 90. *SQL* cannot be used to analyze or utilize the chronological order of events, and therefore lacks a tool to deal with one of the most important relationships in *CEP*. Dealing with the specific order of events is done with a pattern language which is integrated into *EQL*.

RTM's *Analyzer* was designed to satisfy a variety of applications through a set of unique advantages [RTM Realtime Monitoring GmbH, 2010]:

Tailored solutions
: Its "library approach" consists of basic, complementary modules with well-established standard functionality which are coupled, configured and combined with the *CEP* platform to achieve a tailored and lean solution.

Easy to extend
: New functionality and modules are easily integrated.

High software quality
: Highly generic and parameterized modules allow perfect adaption.

Short time-to-market
: Solutions based on existing modules reduce development cycles.

Easy integration
: The Java based modules can be integrated easily into a system landscape.

A new solution using this *CEP* engine has to undergo three steps.

1. Plug-in the incoming streams of events.
2. Define the business logic that analyzes the streams.

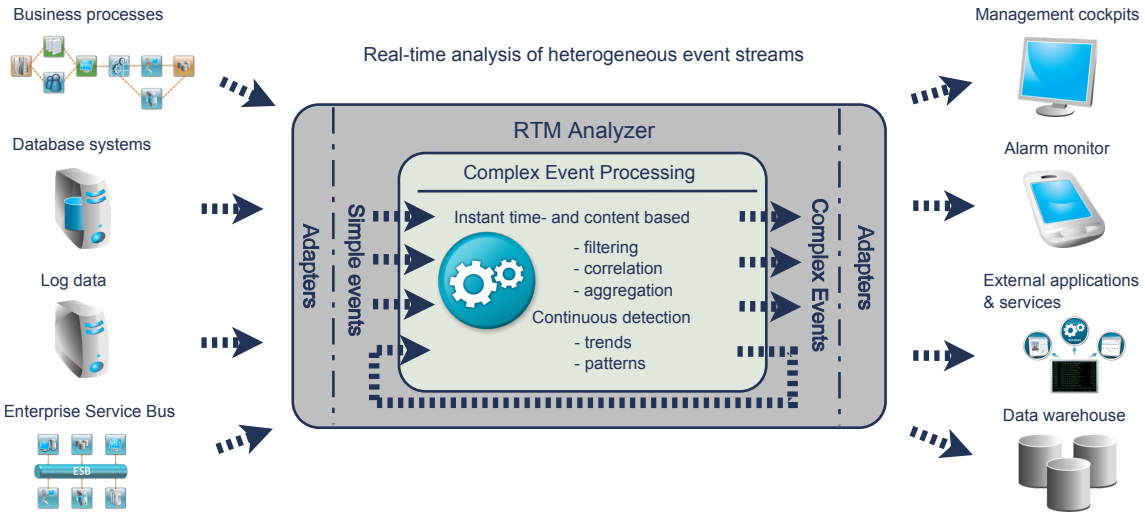

Figure 2.4: Architecture of the *RTM Analyzer* [RTM Realtime Monitoring GmbH, 2010].

3. Plug-in a target that uses the resulting complex events.

Figure 2.4 on page 12 illustrates how the *RTM Analyzer* works. At the left, incoming events are collected from different kinds of sources. *This thesis will show how to collect data from the event bus and from databases.* After selection and configuration of the adapter, the events are ready for analysis. Complex events themselves can serve as the input for further event processing and create higher level complex events. The resulting events are reentered onto the event bus where they can be used by several applications, services and GUIs with the help of adapters.

The *RTM Analyzer* uses an interface based on *SQL* to analyze simple events and to create complex events. Defined queries are continuously run on incoming data. This is fundamentally different from a database system because it is the data that is transient, while the queries are stored. Technically, all the functionality described thus far could, with some restrictions, also be achieved with a database system. However, this would make the database the bottleneck of a bigger system. Both periodic query execution and triggers are designed for static data-sets. Highly dynamic and potentially unbounded event streams can cause a high workload for database systems. The same is true for a large number of triggers.

Perhaps the most important argument in favor of *CEP* in a comparison with database systems in an event oriented architecture, however, is the fact that database systems do not provide the necessary tools to automatically consider time in their computation of events.

In this architecture the producers of events are completely decoupled from the consumer of the same type of events. There might not even be a counterpart to either the producer or the consumer and no element is aware of the existence of the other.

The event bus, which feeds the *RTM Analyzer*, receives its messages via a message oriented middleware (*MOM*). Here the *MOM* is realized in Java Message Service (*JMS*) where a producer registers itself as a publisher for a certain topic – a stream of events – and a consumer registers itself as a subscriber.

In a purely event driven architecture, sensors or other event generators are registered as publishers to that event bus for a certain topic. From that moment on, all their messages are available inside the event bus. A subscriber to that topic might have several very different purposes, e.g., it could collect the events and present them to a human user inside a graphical user interface, store them persistently in a database or perform a hardware action.

The *CEP* engine now is an element that listens to the event bus as a subscriber of one or more topics. It analyzes the incoming events on those topics and reinserts complex events as a publisher to the same or another topic. Figure 2.1 on page 6 shows an example of an event bus with some subscribers, a *CEP*, and many publishers.

This *CEP* also supports queues, where a message has only one defined consumer. But this is only an extra functionality for specialized situations and reflects a more traditional way of event handling, which is not explicitly discussed here.

SQL is the choice for the query language. It has the capabilities to formulate and modify business logic and it has a steep learning curve due to its declarative approach. It is extensively used in database systems, which makes it easy to understand for anyone who has already worked on database systems. Two aspects had to be added to *SQL* in order to query events: the notion of time, which is done with sliding windows, and pattern matching.

Two benefits of sliding windows are:

1. Recent data is emphasized. This is common in applications where the most recent data is seen as more relevant.

2. Precise query semantics produce deterministic and reproducible results.

A stream of events that enters the system and finally arrives in the *CEP* can be differentiated into four different types of streams as shown in Figure 2.5 on page 14.

A raw stream is the type of data stream that arrives at the system. Here, an arriving event (e, t) contains a record e and is precisely valid at timestamp t. It is ordered non-decreasingly by the timestamp in relation to the other events in the stream and every record for each incoming event follows the same schema. The timestamp is set by the application that creates the event, not the system that runs *CEP*. The use of system time on events could create different query results based on slight differences in the arrival time at the

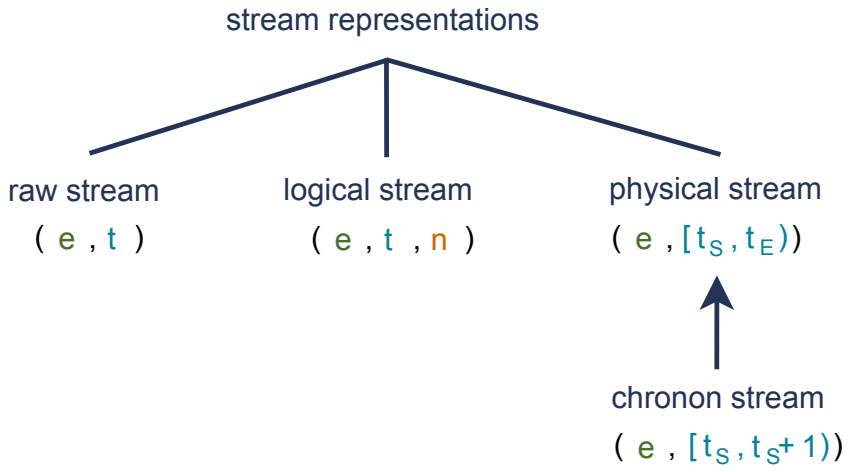

Figure 2.5: Stream representations in *CEP* [RTM Realtime Monitoring GmbH, 2010].

system, so the use of the application time is necessary to create reproducible, well-defined query results.

Conceptually, logical and physical streams are distinguished with their corresponding operator algebra. Logical streams are an order-agnostic multiset representation of either type of stream. Its constituent elements (e, t, n) are extended by the attribute n, which represents the multiplicity of that event. Duplicate elements can be produced at source level or during query processing. Logical streams provide an abstraction from physical-level issues by assuming the availability of all tuples, timestamps and multiplicities. Practically, direct use of a logical stream would cause a huge computational overhead because of the separate evaluations of the operator output at every time instant.

In a physical stream, the events $(e, [t_s, t_E))$ are non-decreasingly ordered by their start timestamps. The stream might contain duplicates but is a more compact representation of the logical view. It also combines identical and consecutive events into a single event with a time interval and orders the resulting stream by start timestamp. A chronon stream is a specific version of a physical stream in which the time intervals are chronons. Chronons, in this case, are the smallest granularity of time that can still be used in *CEP*.

Figure 2.6 on page 15 shows an example in which a physical and a chronon stream are mapped to the same logical stream. All events shown must have the same schema. The letters stand for the different tuples. The event a that starts at t_1 in the chronon stream has three following events with the same tuple and therefore is aggregated into one event in the physical stream. Since at t_2 another event with the tuple a – a duplicate – appears, it is shown as parallel in the chronon and the physical steam. In the logical stream it appears as a 2 at t_2 since the stream does not show duplicates but instead shows the multiplicity of tuples.

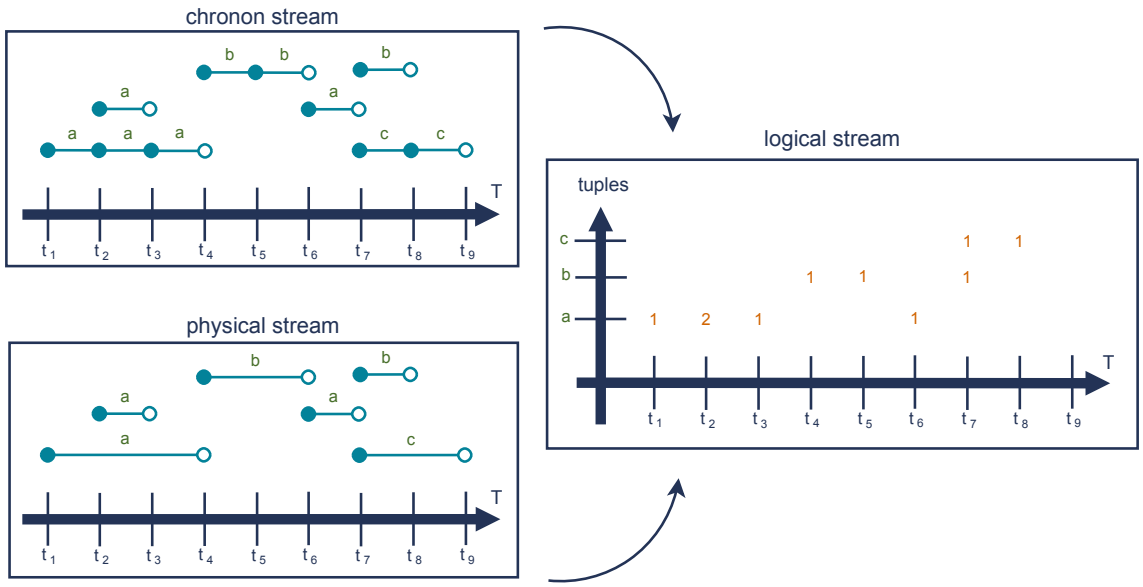

Figure 2.6: Example for stream representation in *CEP* [RTM Realtime Monitoring GmbH, 2010].

2.2 Internet of Things

Traditionally, the internet has concentrated on the user in front of a computer monitor, inputting and outputting data. Approximately 50 petabytes of information is currently available through the internet, and most was generated through typing, scanning or other manual information collection means [Ashton, 2009].

In contrast stands the *Internet of Things*, an idea that most likely originated in a 1999 presentation by Kevin Ashton [Ashton, 2009]. The *Internet of Things* envisions the automation of the data collection by "things" – objects of any size that have some sort of internet access to share their data. *Fraunhofer IML* [ten Hompel, 2011] and *MIT*'s *AUTO-ID labs* [MIT AUTO-ID labs, 2008] have developed *Internet of Things* use cases and scenarios using *RFID* as originators of the data. The *MIT* started the *AUTO-ID* labs with a project designed to equip a company with an *RFID* infrastructure in 1999 [Mattern and Flörkemeier, 2010]. Kevin Ashton was cofounder and director of that project. Fraunhofer IML focuses on the assistance and analysis of logistical processes.

Although much contemporary research focuses on the use of *RFID* as the most important source of information, the *AUTO-ID labs* had a broader view of the *IoT* from its inception, which is indeed a growing reality. Ever since the first iPhone was unveiled in 2007 [Honan, 2007], smartphones have become ubiquitous in everyday life. In an IoT environment, smartphones are both data publishers and subscribers that use non RFID's. For instance, Google collects information from smartphones while they are being used as navigation

devices. Based on this information, Google determines the flow of traffic and openly displays the traffic information in their own online map service [Barth, 2009]. Simply by noting the position and speed of the vehicle, it can conclude whether traffic is flowing at a normal speed or if a traffic jam has occurred. Another example utilizes the three-axis magnetometers that are integrated into smartphones. When placed on the ground, smartphones can actually be used to analyze seismic activity up to a certain level.

The possible examples of use cases for the *Internet of Things* are diverse and numerous. The *iCore* (*Internet Connected Objects for Reconfigurable Ecosystems*) project [iCore Project, 2011a], to which this thesis is intended to contribute, defines four major application domains: *Smart Home*, *Smart City*, *Smart Meeting* and *Smart Business*.

The *IoT* is more of a concept than a precise technology. According to Mattern and Flörkemeier, *IoT* projects utilize different mixtures of the following qualities/abilities [Mattern and Flörkemeier, 2010]:

Communication
: Objects have the ability to communicate via a network, preferably wireless

Addressability
: Objects can be remotely addressed, queried and influenced

Identification
: RFID, NFC, barcodes or other means uniquely identify each object

Sensors
: Sensors collect information about the object's environment

Effectors
: Effectors allow the object to alter its environment

Embedded Data Processing
: Smart objects can interpret their sensor data or store it

Localization
: Objects know their location through sensors or can be localized by others

User Interface
: Smart objects have the ability to communicate with humans

The *IoT* technology is growing rapidly and increasingly appearing in daily life. *RFID*s are used in many places, e.g., a chip with an ID number that identifies employees via reading stations, thus allowing access to an office; or a credit card payment using Near Field

Communication. Its powerful technology requires sensible and responsible use. Unfortunately, poor examples already abound; for example, *Apple Inc.*'s collection of location data from its iPhone customers combined with low security [Williams, 2011]. Therefore, critical discussions and views such as those collected by Rob van Kranenburg [van Kranenburg, 2008], are necessary and valuable.

The *OECD* estimates that by 2020 over 50 billion devices could be online [OECD, 2012]. These devices will be located in a variety of settings; e.g., smart energy devices, smart cars, *eHealth*, and many others. With this scope of information available, data mining will certainly become even more important in the future. The technology around *Complex Event Processing* is very well-suited for this work because of its capacity to analyze massive amounts of streamed data.

2.2.1 The iCore Project

To make better use of the *IoT* and to contribute to the European strategy to "deliver sustainable economic and social benefits from a digital single market based on fast and ultrafast internet and interoperable applications", the *iCore* project [iCore Project, 2011a] has been started. The project is handled by a consortium of 19 members from eleven countries; eleven industrial partners, five research centers, and three universities [iCore Project, 2011b]. This section is based on the official releases of the *iCore* Project [iCore Project, 2011a].

The two major issues targeted by this project are:

a) Abstraction of technological heterogeneity while still enhancing context-awareness, reliability and energy-efficiency.

b) Respecting the views of different users/stakeholders (owners of objects & communication means) for ensuring proper application provision and business integrity, thus maximizing exploitation opportunities.

The *iCore* project proposes three levels of functionality in their cognitive framework to achieve reusability in a high number of diverse applications:

i) Virtual objects (*VO*s)

ii) Composite virtual objects (*CVO*s)

iii) Functional blocks for representing the user/stakeholder perspectives.

Real-world objects are represented in the framework as *VO*s, so they are cognitive virtual descriptions of them. The real-world objects can be sensors, objects or other devices. *CVO*s combine semantically interoperable *VO*s into a single object. An example could be that all the sensors built into an automobile are seen as *VO*s, and some or all of them are integrated into several *CVO*s such as the engine, the cabin and on a higher level even the entire car. The exact definition of *CVO*s and the level three functional blocks depend on the user/stakeholder perspective. An owner of just one or two vehicles has a different perspective than an owner of an entire fleet of delivery vehicles. The technical heterogeneity of a big system resolves into *VO*s and *CVO*s, which hide the low level information and consolidate it into a useful form for higher level services. Either level offers mechanisms with which the services can be registered, looked up, discovered and composed, realizing an open network architecture containing highly intelligent software. Security issues along with data privacy and access restrictions will also be covered by the *iCore* project. The project especially targets the use cases of ambient assisted living, smart office, smart transportation, and supply chain management.

Smart Home

The "Smart Home" domain, as defined by the *iCore* project [iCore Project, 2011a], aims to provide value-added services to the user in his or her connected home through the virtual objects. In the personalized environments of smart homes, services will be automatically created and will have easy-to-use interfaces. In this way, services spanning diverse business sectors such as *eHealth*, security and energy may be provided. Composite virtual objects will stipulate that different business sectors and individuals work cooperatively to maximize not only the individual benefits but also the benefits of the entire system. Smart Home is where the digital and physical objects merge seamlessly.

Smart City

On a higher level, the *iCore* project [iCore Project, 2011a] defines the "Smart City". On a large scale the topics of traffic, tourism, culture, etc. are handled based on the incoming events. The *VO*s represent various domains; e.g., road conditions covering sensors, air quality measuring sensors, vehicle positioning, movie ticket sales, etc.. Aggregated, this data can help identify the opportunistic preferences of specific visitors and predict items of interest. User-specific data is also combined with larger scale data. This can then assist in suggesting places to go, movies to see, traffic jams to avoid, and many more. Combining the data from different domains enriches the output aggregation.

Smart Meeting

The third domain defined by the *iCore* project [iCore Project, 2011a] is the "Smart Meeting". This domain focuses on connecting users and their mobile devices to networks as they are encountered. Information is represented by *VO*s, to facilitate the management of

access rights. The composition of information from different sectors is combined to a *CVO* that is shown to a specific participant, based on the participant's interests and privacy settings. Intelligent functionality presents opportunistically present *VO*s, and autonomously manages its service levels.

Smart Business

The last domain described by the *iCore* project is "Smart Business". It is meant to allow various stakeholders to work together with increased functionality while individually remaining secure. *VO*s are reused throughout the entire supply chain while their opportunistic presence is always being tracked. Alerts are created and sent to stakeholders whenever cognitive technology predicts that predefined parameters are not going to be met within an individual step of the supply chain. Sensors in every aspect of a business, whether inside a warehouse or a printing device, enable thorough analysis of the entire business, while at the same time remaining aggregated into sector-specific groups.

2.3 Example Cognitive Algorithms

For the purpose of demonstrating how and in what forms the *CEP* is capable of integrating machine learning algorithms, two algorithms have been selected for this thesis that reflect real world situations.

First, the *Slope One* algorithm is presented. This is a recommender machine learning algorithm that is realized in the open source machine learning project *Apache Mahout*. A recommender is an algorithm that filters a list of elements based on their content to predict a preference, and returns a recommendation. The example used by the *Apache Mahout* to demonstrate its capabilities is that of a movie recommender; it makes recommendations for movies to see based on the movies already seen by the user and other users. This fits into the *Smart City* domain as defined by the *iCore* project.

The second algorithm presented is the *Fire Detection Based on Intelligent Data Fusion Technology* from Bao et al. [Bao et al., 2003]. It is meant to increase the precision of fire detectors by combining the data from three different kinds of sensors. This is directly applicable in the fields of *Smart Home*, and *Smart Business*.

2.3.1 Slope One Algorithm for Classification

The *Slope One* classification algorithm is one of the algorithms used by the non-distributed recommender engine in *Apache Mahout*. *Apache Mahout* is a framework of machine learning algorithms that is managed by the *Apache Mahout* project of the Apache Software

Foundation. The algorithms implemented in this framework are numerous and are categorized into classification, clustering, pattern mining, regression, dimension reduction, evolutionary algorithms, recommenders / collaborative filtering, vector similarity and others. Almost all of these algorithms use the map reduce *Apache Hadoop* framework and therefore run in a distributed environment. The *Apache Hadoop* framework is another project of the Apache Software Foundation and is an implementation of Google's MapReduce algorithm. MapReduce is used to split up algorithms into their synchronous and parallel elements and to spread their workload and data in a highly distributed environment.

Almost all the algorithms realized in *Apache Mahout* use *Apache Hadoop*, except for the recommenders of the former Taste project which includes the *Slope One* algorithm.

Given a predefined set of items, a list of users, and a non-complete list of ratings for many users of the items, the algorithm calculates predictions on how a given user could rate items the he or she hasn't rated yet. Modified algorithms also integrate more data to determine a value for item similarity or user similarity to integrate into the calculations, possibly enhancing the results. The algorithm shown here was described in the paper by Lemire and MacLachlan [Lemire and MacLachlan, 2005].

The algorithm is split into two major parts. The first part calculates the average difference of ratings between any two items in the data set. The training set χ is the base to calculate the deviation of any two items j and i in $dev_{j,i}$. For every user u who has rated both items i and j, the difference is aggregated and divided by the number of users who have rated both. This makes it the average difference in the rating of the two items. It is shown as the item deviation in equation 2.1.

$$dev_{j,i} = \sum_{u \in S_j, i(\chi)} \frac{u_j - u_i}{card(S_{j,i}(\chi))} \tag{2.1}$$

The second part of the algorithm is the prediction of any item j for a given user u, given that the user has not rated that item yet. To predict the rating user u would give item j, every rating u_i from that given user is added to the deviation of j to i and averaged by the number of rated items. R is the set of all relevant items and is defined as $R_j = \{i | i \in S(u), i \neq j, card(S_{j,i}(\chi)) > 0\}$. The resulting prediction is shown in equation 2.2.

$$P(u)_j = \frac{1}{card(R_j)} \sum_{i \in R_j} (dev_{j,i} + u_i) \tag{2.2}$$

The *Slope One* algorithm is a simplification of that formula. Given the approximation that with a dense enough data set with $card(S_{j,i}(\chi)) > 0$ for almost all i and j, the

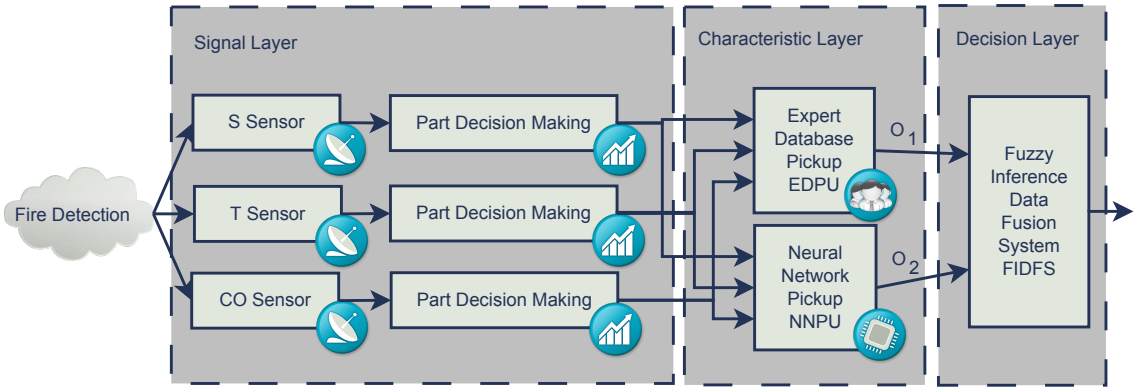

Figure 2.7: Schema of Bao et al. fire detection system [Bao et al., 2003].

algorithm can be simplified by using $\bar{u} = \sum_{i \in S(u)} \frac{u_i}{card(S(u))} \backsimeq \sum_{i \in R} \frac{u_i}{card(S(u))}$. Equation 2.3 shows the resulting *Slope One* algorithm to calculate a prediction for user u on item j.

$$p^{S1}(u)_j = \bar{u} + \frac{1}{card(R_j)} \sum_{i \in R_j} dev_{j,i} \tag{2.3}$$

2.3.2 Fire Detection Based on Intelligent Data Fusion Technology

In 2003, Bao et al. published a paper about their algorithm to detect fire, based on intelligent data fusion [Bao et al., 2003]. This algorithm accumulates and aggregates signals from three different kinds of sensors to increase precision in fire detection while avoiding false alarms. Photoelectric smoke detectors detect particles in the air, carbon monoxide detectors analyze the level of carbon monoxide in the air and temperature sensors register the heat surrounding the sensor. Combined, all of these sensors function proficiently in detecting a fire, whereas individually they may be inadequate at detecting a fire and can set off false alarms. Bao et al. give an example where a smoke detector shows a satisfactory performance in detecting either a flaming or smoldering fire but does not detect smoke particles in invisible or black smoke with particles smaller than $0.4\mu m$. Therefore Bao et al. have combined the data received from all three types of sensors and engineered an algorithm to overcome these individual shortcomings.

Bao et al. split their algorithm into three steps: a signal layer, a characteristic layer and a decision layer, as shown in Figure 2.7. Each of these layers, on its own, creates data that can be used to determine a fire. However, the further the signal travels through the system, the more reliable it is. At this point, the assumption is made that a fire signal is treated as having a value between 0.0 and 1.0, according to the values used in the paper.

In the signal layer, the incoming signal receives a first analysis confirming the validity of

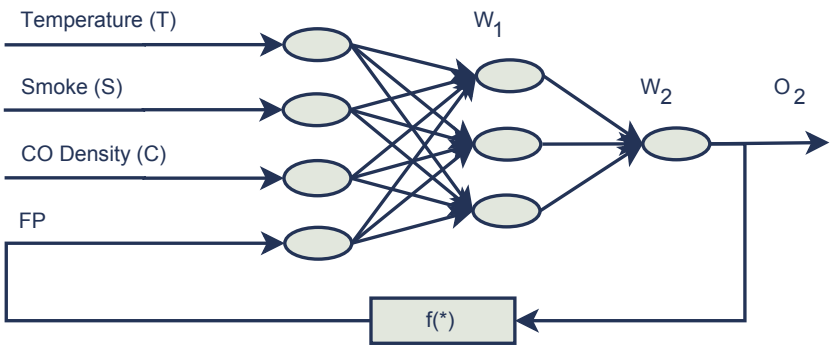

Figure 2.8: Schematic of Bao et al. back propagated neural network for their fire detection system [Bao et al., 2003].

the signal. Instead of simply accepting a high value as a fire signal, the signal is analyzed for a jumping value. The function $a(k)$ of equation 2.4 is defined as the sum of the difference of the k-th sample $X_1(k)$ and the $(k-1)$-th sample $X_i(k-1)$ for the i-th type of sensors. In the output function – equation 2.5 – the value of $a(k)$ is subtracted by the threshold value θ_{lim} and put into a unit step function $U(x)$. It is not explicitly defined which unit step function is used here. From the context of the paper it appears that a function has been chosen that basically rounds the signal value to the first digit after the decimal point. When the function reaches the value 1, for any one k, it can be considered an effective fire signal.

$$a(k) = \sum [X_i(k) - X_i(k-1)] \tag{2.4}$$

$$Y(k) = U[a(k) - \theta_{lim}] \tag{2.5}$$

The characteristic layer is split up into two independent working systems, the *Expert Database Pickup* (*EDPU*) and the *Neural Network Pickup* (*NNPU*). The *EDPU* is a database containing previously defined expert data on the probability of fire given a predefined set of sensor data. Since only eleven values are available, assuming use of the abovementioned unit step, this database contains a maximum of 11^3 entries. However, it is highly probable that there will not be an entry for every given combination. If the sensors' values do not contain a corresponding data set in the database, the *EDPU* is unable to return an estimation on the probability of fire.

The *NNPU* is a back propagated neural network and is capable of analyzing data and returning viable results even on input values that are not predefined (especially if optimally designed and trained). Thus, it has an advantage over the *EDPU*.

The neural network uses four inputs: the data from the three sensors and the tendency feedback signal (FP) which is the back propagated value from the output signal. As long as O_2 – the output of the neural network – fits the requirement from equation 2.6, the feedback as defined in equation 2.7 is valid.

$$\lambda_{j-1} \leq O_2(k) - O_2(k-1) < \lambda_1 \tag{2.6}$$

$$FP(k+1) = \sigma_j + \frac{1}{m} \sum_{t=k-m}^{k} FP(t) \tag{2.7}$$

To train the neural network, the input and correct output from equation 2.8 have been used and iterated 367 times. To do this, Bao et al. have estimated the parameters seen in equation 2.9 based on their experienced analysis and estimates. The results are the two matrices for W_1 and W_2 for the weights of the neural networks seen in equation 2.10 and equation 2.11. The matrices have been tested against real data and have shown a sufficiently small error rate. With these two matrices, the training of the neural network is complete and its full power can be utilized even without further use of the back propagation.

$$[S(k), T(k), C(k), FP(k), O_2(k)]^T = \begin{bmatrix} 0.0 & 0.1 & 0.2 & 0.3 & 0.4 & 0.5 & 0.6 & 0.7 \\ 0.2 & 0.2 & 0.2 & 0.2 & 0.2 & 0.2 & 0.2 & 0.2 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.2 & 0.2 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix}$$
$$\begin{bmatrix} 0.8 & 0.9 & 1.0 & 0.0 & 0.1 & 0.2 & 0.3 & 0.4 & 0.5 & 0.6 & 0.7 & 0.8 & 0.9 & 1.0 \\ 0.2 & 0.2 & 0.2 & 0.2 & 0.2 & 0.2 & 0.2 & 0.2 & 0.1 & 0.1 & 0.5 & 0.5 & 0.8 & 0.7 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.1 & 0.2 & 0.3 & 0.4 & 0.5 & 0.6 & 0.7 & 0.8 & 0.9 & 1.0 \\ 0.2 & 0.2 & 0.2 & 0.2 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.3 & 0.4 & 0.5 & 0.6 & 0.7 & 0.8 & 0.9 & 1.0 \end{bmatrix} \tag{2.8}$$

$$\begin{aligned} j_n &= 20 \\ m &= 4 \\ \lambda &= [0, 0.001, 0.002, 0.003, 0.004, 0.005, 0006, 0.007, \\ &\quad 0.008, 0.009, 0.01, 0.012, 0.014, 0.016, 0.018, 0.02] \\ \sigma &= [0.0019, 0.001, 0.002, 0.003, 0.004, 0.005, 0.006, \\ &\quad 0.007, 0.008, 0.009, 0.013, 0.013, 0.014, 0.013, 0.013, \\ &\quad 0.3, 0.01, 0.02, 0.03, 0.04] \end{aligned} \tag{2.9}$$

$$W_1 = \begin{bmatrix} 2.7998 & -0.1424 & 0.3836 \\ -0.8670 & 2.0763 & 1.1122 \\ -0.2689 & 2.6346 & -0.5426 \end{bmatrix} \tag{2.10}$$

$$W_2 = \begin{bmatrix} 0.5442 & -2.1063 & 0.0204 \end{bmatrix} \tag{2.11}$$

Finally, in the decision layer the *Fuzzy Inference Data Fusion System* (*FIDFS*) is used to reduce the rate of false alarms. The outputs from the *NNPU* and the *EDPU* are fuzzified here as well as subjected to a newly introduced time factor. The time factor will show the duration of time during which the sensor values indicate a fire. It is defined in one of two ways shown in equation 2.12, using T_d as a warning threshold with a suggested value of 0.5.

$$\begin{aligned} T_s(n) &= [T_s(n-1)+1] \cdot U(O(x) - T_d) \\ T_s(n) &= [T_s(n-1)-1] \cdot U(T_d - O(x)) \end{aligned} \tag{2.12}$$

The values from the characteristic layer are fuzzified in three sets and the Gauss function is used as their membership function. The sets are *Positive Big* (*PB*), *Positive Middle* (*PM*) and Positive Small (*PS*). The time factor is only fuzzified into the two sets *PB* and *PS*. Based on their experience with fire, 18 rules were suggested, three of which were stated:

R^1 : IF [O_1 is PS] and [O_2 is PS] and [T is PS] THEN [μ_D is PS]
R^2 : IF [O_1 is PS] and [O_2 is PM] and [T is PS] THEN [μ_D is PS]
...
R^{18} : IF [O_1 is PB] and [O_2 is PB] and [T is PB] THEN [μ_D is PB]

With the above rules, the non-fuzzy fire probability can be calculated using the *MIN-MAX-COA* inference and the weighted-mean de-fuzzified algorithm in the equation 2.13.

$$\begin{aligned} u_D &= \frac{\sum w_i u' D_i}{\sum w_i} \\ w_i &= u_{Ai}(O_1) \wedge u_{Bi}(=_2) \wedge u_{Ci}(T_s) \\ u'_{Di} &= u_{Di}^{-1} \end{aligned} \tag{2.13}$$

2.4 State of the Art

Current systems that incorporate cognitive algorithms and adapt settings based on their collected information are usually standalone systems. They form a closed system, uniting all their components within this single system.

A crawler collects information from various sources. These sources can be sensors, databases, text documents or other locations. The collected information is stored in a persistent place within the application.

A decision unit then analyzes this data in periodic runs and generates an action which is executed by an external component. In most cases, this analysis is run once every hour or even less frequently. Systems such as this are often part of business prediction engines that run overnight due to the large volume of data and calculations.

Software AG's Optimize is one such system which contains a prediction engine to forecast upcoming events. *Optimize* is a *Business Activity Monitoring* (*BAM*) software used to optimize business processes. *Optimize* already uses events as the source for its analytic engine, but the cognitive algorithm in the prediction engine is decoupled from events.

In general this system collects all information itself and directly triggers the actions based on its data.

Two major disadvantages resulting from this approach are:

Lack of Scalability
A single system can easily be overloaded.

Slow Reaction Time
Before a reaction to incoming data is possible, information must first be collected by a crawler. The algorithm picks up this piece of data on its next run.

The prediction engine is directly responsible for collecting and handling the information of all the sensors; therefore creating a bottleneck as seen in Figure 2.10 on page 26.

2.5 Summary

This chapter provides background information about concepts that are foundational to the following sections of this thesis. First, *Complex Event Processing* is presented. The next concept, the *Internet of Things*, is first described generally, and then specifically as seen by the *iCore* project. Last, two examples of cognitive algorithms with practical applications in an IoT environment are presented in full

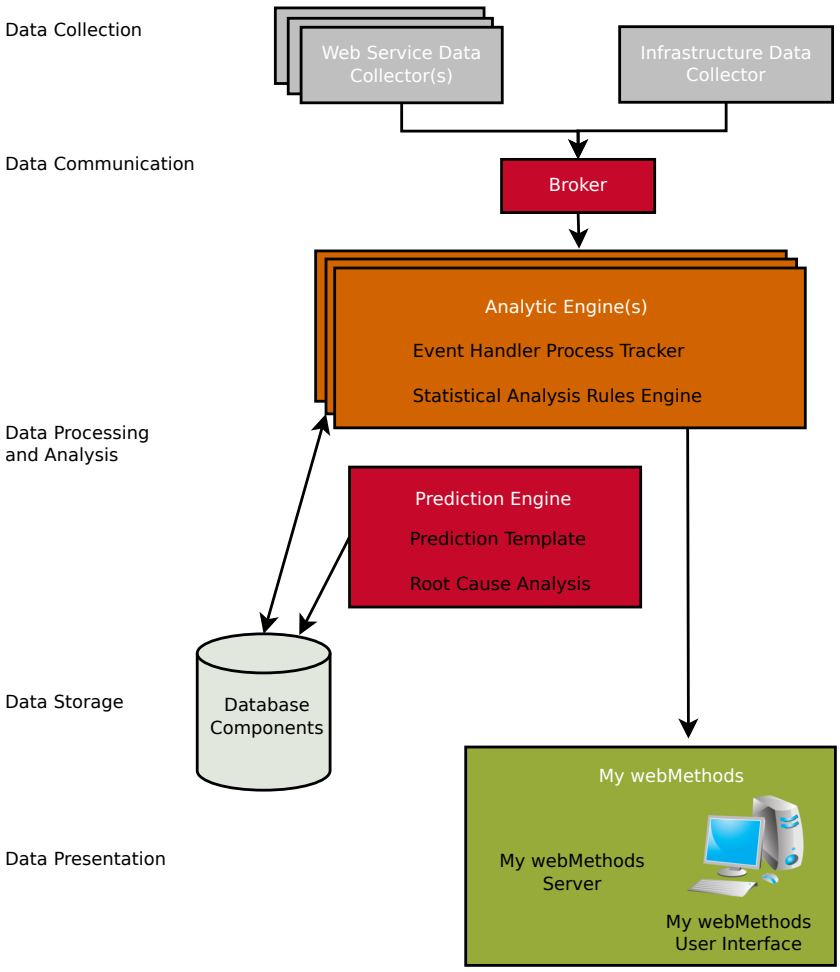

Figure 2.9: Typical use of *Software AG's Optimize* [Software AG, 2011].

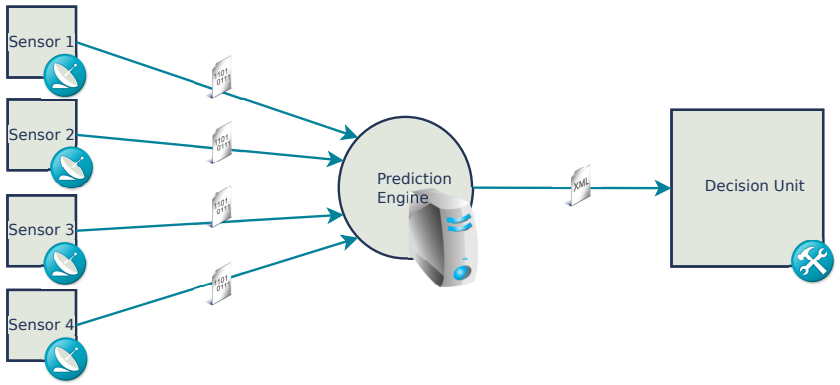

Figure 2.10: Original design of cognitive systems which collect data from sensors and process the data

3 CONCEPTS FOR THE REALIZATION OF COGNITIVE ALGORITHMS

The field of cognitive technology provides a wide range of possible algorithms and scenarios. *Software AG's CEP* provides a set of tools to analyze data in both large amounts and in real-time. This chapter will discuss generally how cognitive algorithms are used and realized, using *Software AG's Optimize* as a specific example. It will also discuss how *CEP* can be utilized to integrate common cognitive algorithms. Each function is presented and analyzed before proposing a method for selecting the optimal function.

3.1 Original

Typically, cognitive algorithms are used in a batch environment where data is collected, stored and then periodically queried. An external unit makes decisions based on the results of the cognitive algorithm, which it must request. Figure 3.1 on page 28 shows this typical scenario and Figure 2.9 on page 26 shows a real life example of how *Software AG's Optimize* works. Offline algorithms are more likely to operate in this way than online algorithms since they process all available data at once.

Generally a cognitive application contains the following five components:

Data Source
: The source of data for a cognitive application can have various shapes and primary uses. It can be a database, a webpage, a sensor, or an event. The source does not have to be aware of the use inside the system, nor does it have to be designed for the system. The source remains passive in regards to the system.

Crawler
: The crawler collects the data from the data source and stores it in the database. This is the first active part of the system and it is specifically designed for the data source it crawls. If a system uses a series of data sources, several crawlers could become necessary, with each specialized for its particular type of data source.

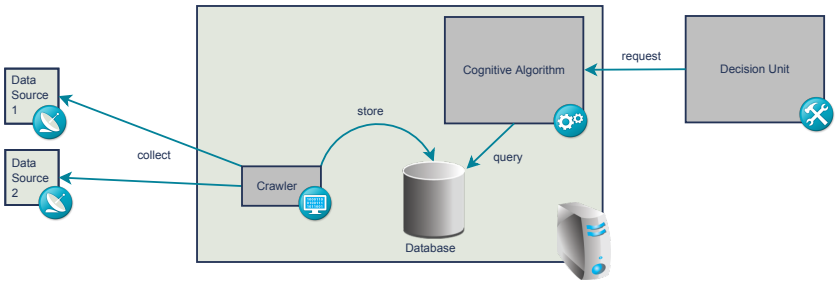

Figure 3.1: The typical way to realize machine learning.

Database

The database, or data storage, is the passive unit that holds the data collected by the crawlers and provides the data to the cognitive algorithm. The realization of this component depends on several criteria; the system, the data size and layout, and the need for speed or richness of data. Generic *SQL* databases are easy to integrate and handle. They can hold a lot of similarly-structured data, but might lack the speed needed for some operations while being too detailed for other applications. In general the database is passive; it receives the data from the crawlers and queries from the cognitive algorithm. In some cases it could be advantageous to build triggering mechanisms into the database, allowing it to take over some of the CPU cycles.

Cognitive Algorithm

The heart of the application is the cognitive algorithm itself. Depending on the usage, it could be triggered by one of three methods: a timer, the crawler indicating that it has finished, or an outside request to run. The algorithm queries the database for required data and then calculates the output, either providing its data to the decision unit or by writing it back into data storage – either in the same database or a different one.

Decision Unit

The decision unit may or may not be part of the cognitive system. In most cases, it will be an external component that requests the output from the cognitive system and makes decisions based on the output.

The structure described here is a general layout scheme for the way a cognitive application will process data from its source to the decision unit. None of the parts have to stand exclusively. Many crawlers can collect data from various sources and store it in various storage locations at the same time depending on its intended usage. Many different cognitive algorithms could run at the same time allowing comparison between them or allowing a more reliable mean value of the results. The variations and usages are quite diverse.

The examples Harrington outlines in his book do not use specific crawlers because the data already exists in a format the algorithm can use to process it.

The two real-life examples previously mentioned also work in this manner. The *Slope One* algorithm described in section 2.3.1, which is also realized in *Apache Mahout*'s recommender as the standard algorithm, works just this way. Here, the data is prepared for usage and loaded into the memory of an object of a data storage class, then the algorithm queries this object. This class can get very large and may require manual expansion of heap space in order for the application to run properly. This class is specialized for a type of input data which increases the speed during the query stage. The *Fire Detection System Based on Intelligent Data Fusion Technology* algorithm from section 2.3.2 also operates this way, but is more complex. It works in three steps as shown in Figure 2.7 on page 21. The output of the previous step is used as the input for the next step. Three crawlers individually collect the data from the three types of sensors and perform a first analysis of the incoming data. An exact description of how the data is stored is not provided. Then the *EDPU* and *NNPU* process the results in parallel. In this case, the *EDPU* possesses an internal database containing the expert data. The third section could either be viewed as another layer for a new cognitive algorithm or as a decision unit which already contains a cognitive algorithm.

A different method of storing and processing the data is to distribute it over a network and have every node take over some of the computation. Most of the algorithms included in the *Apache Foundation Mahout* framework use the *Apache Hadoop* framework to distribute both data and processing to enable the scalable processing of large data sets. The *Apache Software Foundation Hadoop* project consists of an implementation of *Google*'s *MapReduce* algorithm, and components that support the execution of the algorithm such as the utilities package and the distributed file system *HDFS*. Distribution of the computation using *Apache Hadoop* spreads the data over all nodes involved in the process and the processing of the algorithm is performed on all nodes as well. The distribution of both code and data is handled by the framework.

This common way of handling cognition is computationally expensive and rather static. *Apache Mahout*'s motivation to use *Apache Hadoop* is based on distributing the work load of such expensive algorithms. The time required to compute complex algorithms has led to creative, clever, and effective ways to reduce the number of calculations required in some cases, e.g., the *Apriori Principle*. These algorithms work very statically and rely on a request to collect data and run.

3.2 Relational Databases

The typical uses of cognitive algorithms – as described in section 3.1 – utilize databases that are solely used for data storage and are only queried to feed the cognitive algorithm,

as in Figure 3.1 on page 28. This section discusses the usage of databases for cognition where queries and a series of views realize the cognitive algorithm. This way of realizing a cognitive algorithm is related to the variant utilizing *CEP* by the query language.

Compared to the system depicted in Figure 3.1 on page 28, the system described here combines the cognitive algorithm and the database into a single system. The other components remain the same and are not mentioned here.

Many algorithms can be split up into several steps from which some can be translated into *SQL* queries. Problems can occur when individual steps consist of complex structures or hierarchical data (e.g., trees in *Decision Trees*, the follow up graph in the *Apriori Association Analysis*, or the neural network in the *NNPU*). This is based on the fact that *SQL* is not computational complete [Atkinson et al., 1995]. Although not impossible to realize in this manner, the complexity could easily exceed a feasible size.

SQL databases only allow simple data types and any relation between them has to be realized with auxiliary tables or columns. While an object oriented language is able to handle these relationships with ease, a relational database requires a substantial effort to realize them. Examples from Lepekhin [Lepekhin, 2004] and Volk [Volk, 2002] demonstrate the complexity of realizing trees in relational databases.

The following list shows the functionality within many relational *DBMS*s that can help to deal with higher complexity:

Views
: Views can assist in distributing the complexity of an algorithm into several levels, but can waste computer cycles by running redundant or nearly redundant queries. The use of auxiliary tables can reduce the waste of redundant calculations by storing derivative data.

Triggers
: Triggers can assist in starting queries or procedures that otherwise would have to wait for a request by the user, a periodic timer, or a third component to start. On the other hand, triggers can bring a system to a halt when the triggering event happens too often, causing the trigger to call a query or procedure repeatedly in a short period of time.

Procedural Database Languages
: Modern DBMSs have several mechanisms that allow the pure set oriented *SQL* to be enhanced with a procedural database language, e.g., *PL/SQL* in *Oracle*, *Stored Procedure Language* (*SPL*) in *Informix*, or *PL/pgSQL* in *PostgreSQL*. These languages can assist in realizing the algorithms, enhancing speed, and optimizing the DBMSs to realize cognition. The procedural blocks of computation can be initiated by either a query or by a trigger.

For use in cognitive algorithms, views are very likely to be too slow. This is because every view needs to be processed every time a query that uses it is started. Redundant and CPU intensive calculations are repeated for every call. Views are only invoked manually, and when combined in another query actually increase the time needed for computation exactly when results are required. Triggers can help realize a system that keeps its data consistent. A *DSMS* may be simulated with triggers up to the point where output is created, but they are generally too slow to deal with a high frequency of events. Procedural database languages are an important addition to *DBMS*s that broaden their possibilities. This effect is also evident in *Complex Event Processing*.

3.3 CEP Queries

CEP queries, written in *EQL*, are the base of all *CEP* in *Software AG's CEP*. Further steps may involve database integration into the *CEP User-Defined Function*s or *Operators*, but everything starts with the *CEP* query.

The *Event Query Language* (*EQL*) contains every query element that is available in standard *SQL*.

The structure of the data is defined in *EDA* files in *XML*. Data manipulation does not occur in the same way as in *SQL* databases. Insertion of events onto the event bus happens via infrastructure adapters, and the re-entry of complex events through a query execution. Re-entered results from a query do not have to be published in the same topic; in fact, it is more likely that they would be published under a different topic. Data manipulation, such as changing events from their original version and overwriting them, does not occur. Rights to read or write a topic are not handled via *SQL*'s *DCL*, but in a command line application "jmsadmin".

The full grammar of *EQL* is located in the Appendix in section 5.4 on page 90. Not all elements shown in this grammar have been realized yet, but *Software AG* has plans in place to do so. The elements that are not included are sub-selects, *NOT IN*s using query results rather than only sets, *OUTER JOIN*s, *EXIST*s, *LIMIT*, and *SORT*, among others.

Any cognitive algorithm that can be realized solely using *SQL* queries can also be realized in *EQL*, unless it uses one of the not yet realized elements where they cannot be replaced with substitutional code constructs.

A cognitive algorithm is processed and executed in *Software AG's CEP* as shown in Figure 3.2 on page 32. Some sensors, or any other source of events, publish events to the event bus. Hereby, the events are published under one or many "topic/s", following an event type definition determining which elements exist and the data type they possess. A *CEP* query inside the *CEP* engine analyzes incoming events and produces complex events

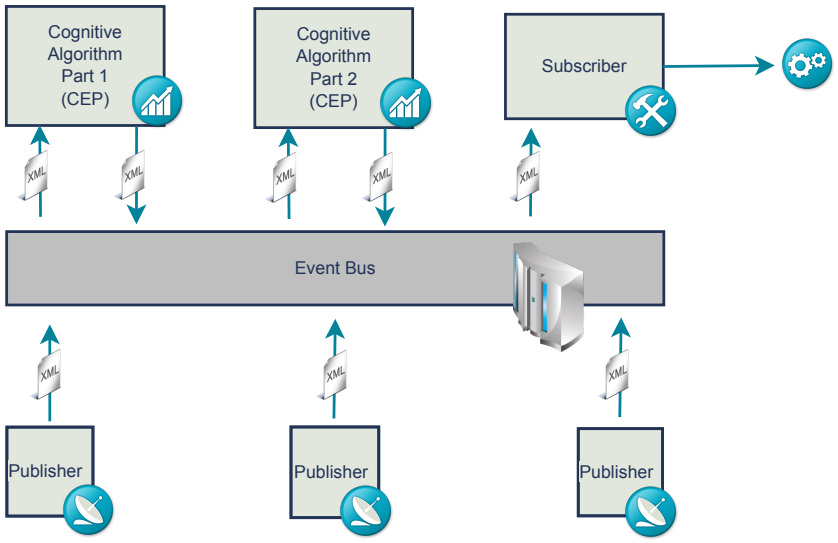

Figure 3.2: Realization of a cognitive algorithm using only a *CEP* query.

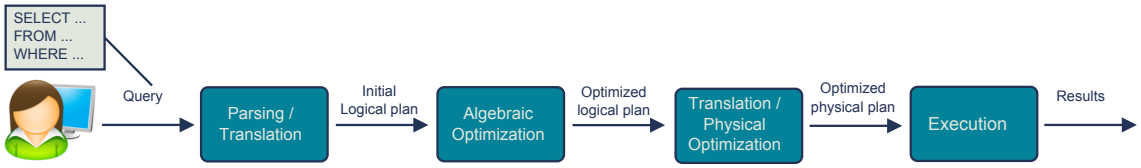

Figure 3.3: Steps in *EQL* query processing [RTM Realtime Monitoring GmbH, 2010].

that are published back onto the event bus. In this case, the *CEP* query is a subscriber to a topic and a publisher to another topic. There may be a parallel, sequential or other type of complex setup of more than just one *CEP* query working on that event bus. Some algorithms simply will not be able to be run in a single *CEP* query. Often they must be split into many sequential operating queries. One or more subscribers receive events and do something with them, e.g., determine actions to be taken, store them to a database, present them on a *GUI*, etc.

The relationship between *SQL* and *EQL* can also be seen when a query is processed, as shown in Figure 3.3. With *EQL*, the query runs continuously and has to respect time aspects, as opposed to *SQL*.

A pure *CEP* query encounters the same or slightly greater difficulties as a *SQL* query in coping with complex relationships. The data types handled in *CEP* are limited to byte, short, integer, long, float, double, character, and string. The topics which act as time variant tables in *CEP* possess a structure which, although designed as a tree structure in *XML*, acts merely as a sequential list of columns in a *SQL* table.

CEP queries can make use of some standard functions to perform some basic mathemati-

cal calculations instantaneously. Besides these standard functions, *User-Defined Function*s can be used to integrate special functionality, which will be described later. To demonstrate the power of EQL, the standard functions are listed here:

ABS
Calculate the absolute value from a number.

CAST
Cast a value to another data type.

CEILING
Determine the smallest integer value that is larger or equal than the input value.

CHARACTER_LENGTH
Determine the length of a string.

EXP
Obtain the power x of *Euler*'s number.

FLOOR
Generate the biggest integer value that is less than or equal to the input value.

LN
Calculate the natural logarithm of the input value.

MODULO
Obtain the remainder of a division of the first input value by the second input value.

POWER
Compute the value of the first input value raised to the power of the second input value.

ROUND
Round a number to a given number of decimal places

SQRT
Apply the square root of the input value.

TRUNCATE
This cuts off any decimal behind the given number of decimal places.

Having compared the topics to tables in a relational database, it now remains to describe how an *SQL*-like language created for relational algebra can work on streams. When a set of inputs $S_1, \ldots S_n$ is entered and an operation op_S is executed on that input, the output

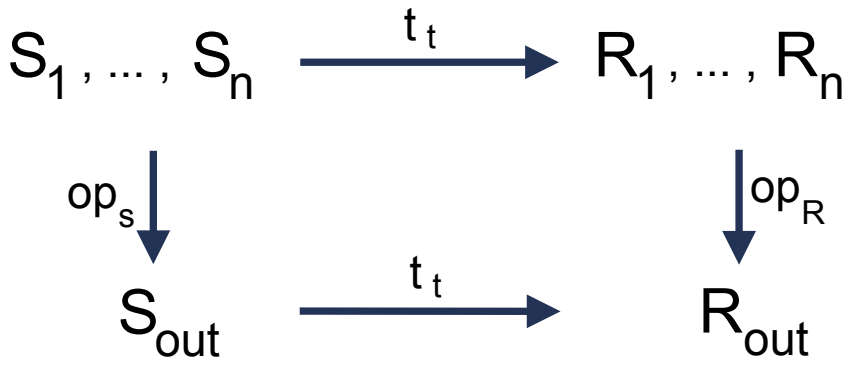

Figure 3.4: Snapshot reducibility as basis for relational algebra in *CEP* [RTM Realtime Monitoring GmbH, 2010].

S_{out} is generated. The relational counterpart is an operation op_R that is executed on inputs $R_1, \ldots, R_n$ which creates the output R_{out}. A *CEP* system is snapshot reducible when the same result as its relational counterpart, as shown isFigure 3.4 on page 34, is returned at every time instant t. If a system is snapshot reducible, the results of the queries are consistent and reproducible.

Time is a very important component in *CEP*. An event in a chronon event stream has a time interval of 1 ms: the smallest time interval still observable in *CEP*. Queries only access the events that are currently visible, whose start timestamp has passed and the end timestamp lies in the future. Because of this, running queries on a topic would only work on a limited amount of events created at the same time. To allow a query to capture more events from a wider time interval, a window can be used which is declared in the *FROM* clause of the query after the topic.

A window that allows a query to access the events of the last five minutes would be declared as *WINDOW(RANGE 5 MINUTES)*. With each incoming event, the events from the last five minutes are queried.

To reduce the amount of times a query is executed and to increase performance, a window can receive a slide time that prompts the query only to be executed after a given interval. A sliding window could be declared as *WINDOW (RANGE 10 MINUTES SLIDE 5 MINUTES)*. This causes a query only to be executed every five minutes, but then access the events from the past ten minutes.

Two special cases are described below:

Range = Slide
: A *tumbling window* has a maximum slide value compared to its window size, without leaving a gap between two sequential windows. It periodically collects the events

without overlapping or skipping any events. An example for this is *WINDOW (RANGE 10 MINUTES SLIDE 10 MINUTES)*, where a query is executed every ten minutes, accessing all events from the past ten minutes.

Range < Slide

A *gap window* neglects events that fall between two specific sequential *RANGE*s, while it is still sliding. This affects a proportion of all time, as is calculated using $\frac{SLIDE-RANGE}{SLIDE}$. In the window defined as *WINDOW (RANGE 10 MINUTES SLIDE 30 MINUTES)*, the query would execute every thirty minutes using the last ten minutes of events. In this case, in $\frac{30-10}{30} = \frac{2}{3}$ of the time, the incoming events would be neglected.

Every incoming event's end timestamp is assigned the time window – the part declared with the keyword *RANGE*. In other words, it is added to the event to keep it available for that time ($t_E = t_s + w$). This does not affect parallel operating queries.

Additionally, an item-based window exists that collects a certain number of successive events before execution. The notion of a slide changes here: it also refers to the number of successive events. A window that collects ten events and jumps to every fifth event would be declared as *WINDOW(ROWS 10 SLIDE 5)*.

The concept of a continuous stream of events and of window based queries is problematic with most offline cognitive algorithms. To achieve optimal precision, the algorithms use all available training data and run the algorithm iteratively. The higher the number of runs, the more reliable the resulting values will be. (This pertains mostly to the weights of the algorithm.) A *CEP* query is executed whenever a new event or a heartbeat (a specific type of event without data and with the sole purpose to prompt execution) arrives at the event bus. This demonstrates that a new execution of the query will contain new data, and therefore has different preconditions than the prior execution which may produce unexpected results. The repetitive iteration is intended to increase precision on a static training data set.

This leaves basically the following options:

a) Only run one iteration.

b) Perform the training outside of *CEP*.

c) Run repetitive learning with tumbling windows.

The last example only functions with unsupervised algorithms. Online algorithms on the other hand, can deal very well with continuously incoming events. However, performance problems may occur if a complex algorithm is forced to run with every new incoming event.

For most cases of supervised learning, the training itself would function better if executed outside *CEP*.

The *k-Nearest Neighbor* algorithm consists of two steps. In the first step, the calculation of the distances from the incoming event's element, pertaining to every element in the training set, can be easily achieved. The second step determines the class that occurs most within the k closest elements. Without sorting, the k best results cannot be determined. A *CEP* query requires extensions to be capable of calculating this algorithm.

EQL also encounters problems, just as *SQL* does, with mapping trees in the data, which makes *Decision Trees* less useable. The already complex solutions shown for *SQL* cannot easily be transferred into *CEP* due to the time aspect. Auxiliary tables can only be realized as auxiliary topics inside the *CEP*, but with time, the events published on the event bus will vanish while "overwritten" events still persist. The Shannon entropy can be calculated in a query, but once again, it cannot be determined which solution is the better one, due to the lack of a sorting mechanism.

The *Naïve Bayes* encounters the same problem. The probability of the different classes can be calculated within a query, but the lack of sorting makes it impossible to determine the best solution.

The *Logistic Regression* uses a step function that gradually increases its returned value between the input values, from -5 to $+5$. Such a function cannot be realized in *CEP* and would need to be replaced with a simpler step function, such as the Heaviside function. The weights are calculated using gradient descent. These functions can be translated into *SQL* queries, but the aforementioned problem remains: that those values are difficult to store once calculated. When the learning is executed externally, the weights can easily be integrated into a *CEP* query.

The same problem occurs in *Support Vector Machines*. The distance from any element to the hyperplane can be calculated, as well as the optimization, but to determine the best value the sort mechanism is required.

The *Linear Regression* also consists of equations that can be calculated in queries, but the *CEP* lacks the sorting mechanisms to determine the best result and a persistent storage for the calculated weights.

The *k-Means clustering* encounters the same problem. Checked events are easily clustered and marked to identify that cluster. However, in order to identify the closest cluster to a new event, a sorting mechanism is required. Each query can utilize both the new incoming event and the resulting events of prior query runs; they can be marked and compared to determine the distance between the mean of the cluster and the new event.

The calculations required for the *Association Analysis* can be executed in *CEP*. Both the *support* and the *confidence* can be calculated for every hierarchical level that is predefined in a separate query, meaning that at each level, the number of items in a set has to be hand written. This makes it impossible, in most cases, to look at all the levels. The realization of the *Apriori Principle* is possible, but again, lacks persistent storage for the lower level results. Once again, it is also a problem to sort the results.

In general, the *CEP* queries are a good starting point for cognitive algorithms since they are the base that collects the events as singles or in windows. They are also capable of calculating most of the required algorithms, but the lack of capacity to sort results makes it almost impossible to complete them.

3.4 CEP Queries Using Database Connectivity

Software AG's CEP allows data to be integrated from a relational database directly into a *CEP* query. Adapters exist for *Apache Derby*, *Microsoft SQL Server* 2000 through 2008, DB2, *Oracle* 8 through 11, and *Sybase* 12 and 15. A single database table can be accessed in the same manner as an event stream, as long at least one real event stream is involved. The database table does not provide a time element as required in *CEP*; hence the time is taken from the involved event streams. Every database table that is accessed requires its own separate configuration.

Once the database access is configured, the table can be used in the query as if it were an event stream, or a regular *SQL* query. Figure 3.5 on page 38 shows that the *CEP* query accesses the database and everything remains the same as it was in the pure *CEP* query scenario.

Since a single query can potentially be executed every millisecond, the access to a relational database by a *JDBC* adapter could quickly turn into a bottleneck for the system. To reduce the number of times the external database is accessed, caching options may be selected.

Three options exist for caching:

No caching
: The database table is accessed whenever the CEP execution needs to access the information in the table.

One-time caching
: The database is only queried once. This is only useful for tables containing some type of immutable master data.

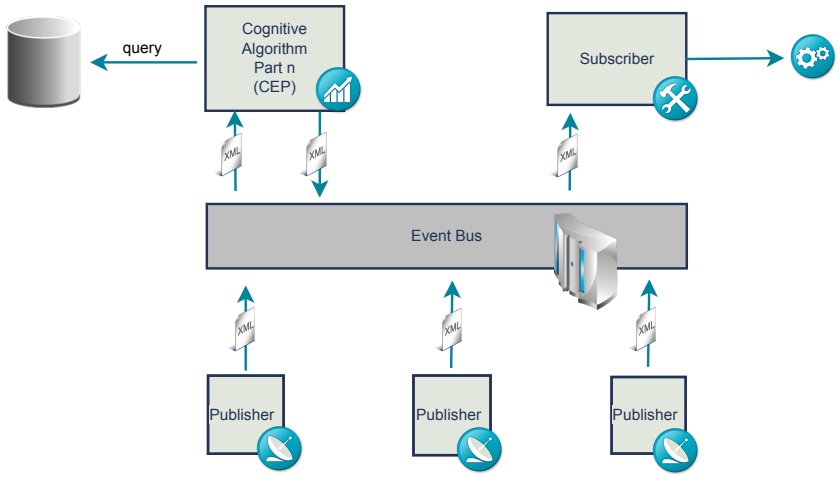

Figure 3.5: Realization of a cognitive algorithm using a *CEP* query with access to a relational database.

Periodic cache refreshing

After a start date, the database table is queried for its contents according to predefined intervals.

Choosing one-time caching or periodic cache refreshing, greatly reduces the number of database accesses, but increases the likelihood that the data accessed in queries may already be outdated, deleted, or replaced.

Generally, there are no existing write methods that are capable of adding entries into a database which will improve static master data. Furthermore, this would not fit into the descriptive approach of *CEP*. An example might be a system in which events only contain IDs, and queries resolve those IDs to some human readable form for the visualization of the complex events of a high hierarchical level. This could assist a *GUI*-layer.

Also, the use of a custom subscriber is possible to update database values as shown in Figure 3.6 on page 39. Any algorithms that store weights or mean values can profit from this approach. A query could calculate the weights to an arbitrary cognitive algorithm in step one with a tumbling window query. In step two, the weights could be collected by a subscriber that updates the weight values in the database in time intervals that are half the size of the window from query one. This would ensure that they are available in the following run.

Back to step one, the query could access the old weights from the last run which are stored in the database and include them in the algorithm that updates the weights. This completes the training part of the algorithm. In step three, a *CEP* query collects the

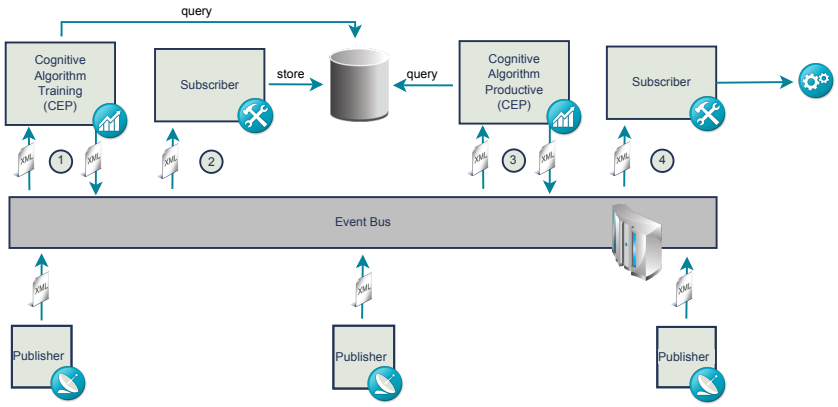

Figure 3.6: *CEP* query with access to a database filled by assistant subscriber.

incoming data – the same data as in step one – and multiplies it with the weights to obtain the results of the algorithm. These results are then used in step four.

Using these methods ensures that many cognitive algorithms previously discussed can be a valuable tool for cases in which the *CEP* lacks a persistent way to store values, making them available for future queries. This includes *Logistic Regression*, *Support Vector Machines*, *Linear Regression*, and *k-Means Clustering*.

3.5 CEP Queries Using Pattern Matching

CEP queries are capable of being extended with pattern matching abilities that can find and analyze patterns in event streams. Pattern matching presents a huge advantage in the processing of event streams over pure *CEP* queries. It creates the capacity to look into the chronological order of events and also to search for patterns in this order. The *RTM Analyzer* tutorial [RTM Realtime Monitoring GmbH, 2010] abstractly defines patterns as "[...] a sequence of elements with element attributes matching certain conditions. Following the *CEP* paradigm these conditions also include temporal relationships [...]". The matcher requires "strictly monotonously increasing start timestamps" to produce output and ignores events in succession which might fit the pattern, but have the same timestamp.

The syntax for pattern matching extends the selected event types in the FROM clause:

```
MATCHING (
    [PARTITION BY <columnList>]
    [MEASURES <name> <type> [DEFAULT <value>]
        (, <name> <type> [DEFAULT <value>])*]
    PATTERN <pattern>
    [DURATION <time>]
```

[WITHIN <time>]
[DEFINE (<patternvariable>
    [AS <predicate>]
    [DO <action> (, <action>)*])*])

This is only the basic syntax for pattern matching: the full grammar can be seen in the Appendix in section 5.4. The main part of the pattern matching follows the keyword *PATTERN*. Here the expected pattern must be defined by using regular expressions. The *PARTITION* clause helps to group the incoming events just as a *GROUP BY* in *SQL* does. The pattern is only then run on events that have the same value in the attribute defined in the *PARTITION* clause. By using *DURATION*, the time can be set after a match becomes valid, in case no further events follow. This is important for finding "non-events". The keyword *WITHIN* works in the opposite manner: in that case, a match is only valid if it is contained within the defined time span.

Inside the *MEASURES* clause, variables can be set which are accessible from the rest of the *EQL* query. The data types for these variables are limited to those available in *EQL* (byte, short, integer, long, float, double, character, string). Inside the *DEFINE* clause, these variables can be set and altered in the *DO* part, and conditions can be checked in the *AS* part.

```
1  SELECT
2    id AS itemID
3  FROM
4    Bid MATCHING (
5      PARTITION BY itemID
6      MEASURES id Integer, currPrice Double
7      PATTERN 'ab{3}'
8      DEFINE
9        a DO id = itemID, currPrice = bid_price
10       b AS bid_price >= 2*currPrice DO currPrice = bid_price
11 );
```

Listing 3.1: Pattern matching example: determine itemID for items with exponential increase in the bid price [RTM Realtime Monitoring GmbH, 2010].

Listing 3.1 shows an example of a query using pattern matching from the *RTM Analyzer* tutorial. The goal of the query is to find the *itemID* of items where the bidding price increases exponentially. The incoming events are partitioned by the attribute *itemID* in line five. The code analyzes a pattern $'ab\{3\}'$, defined in line seven. Therefore a is the first event found of the event type *Bid*. For this item a, both the *itemID* and the bid price are

stored in line nine into variables which were defined in line six. Now three following events are expected, and needed to return a result in this query. Each one must have a bid price which is twice as much as the previous one, as defined after the keyword *AS*. If this is the case, the current price is set to the new event *b*'s event price, as defined after the last *DO*.

```
1  SELECT
2    id AS HighestValue
3  FROM
4    Incoming MATCHING (
5      MEASURES id Integer DEFAULT 0 , currHigh Double DEFAULT 0
6      PATTERN 'ab{200}'
7      DEFINE
8        a DO id = eventID, currHigh = eventValue
9        b AS eventValue >= currHigh DO id = eventID , currHigh =
             eventValue
10 );
```

Listing 3.2: Pattern matching example that determines the event with the highest value.

Pattern matching can even be used to determine a maximum or minimum value of a list of events, as is shown in Listing 3.2. This could be the missing link needed to augment all of the presented algorithms to capacity. Unfortunately, all the algorithms run a preliminary calculation. If this is done in a single *CEP* query, which is very likely, outcoming events can have the same start and end timestamp when windows are being used. As described before, this process eliminates the possibility of using pattern matching on these results. It is worth noting, however, that this is a valid solution for event streams that are guaranteed not to create events with identical timestamps.

The same problem also affects offline algorithms in either of two scenarios. In the first, the order of the events is less important and patterns are seldom of interest. In the second, all of the data is analyzed in a single query, which results in identical timestamps for multiple events. Of all the algorithms presented, only the *Fire Detection System Based on Intelligent Data Fusion Technology* algorithm uses a mechanism that identifies steps in sequential events. In this situation, it can be assured that two successive events will not have an identical timestamp if the sensor's time interval between sending events is greater than the minimum time interval that the *CEP* is capable of handling (1 ms). Additionally, the unit step analysis of the sensor value is the first part of the fire detection algorithm and therefore does not rely on the results of a preliminary query, which would again generate identical timestamps.

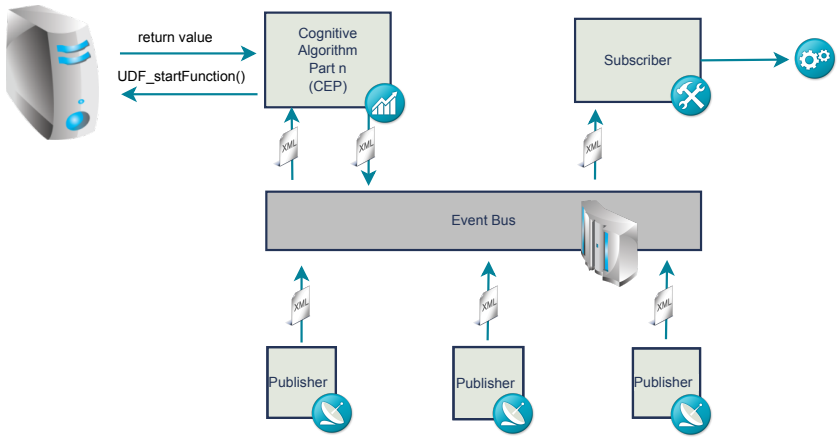

Figure 3.7: Realization of a cognitive algorithm using a *CEP* query utilizing a *UDF*.

3.6 CEP Queries Using User-Defined Functions

The *User-Defined Function* (*UDF*) is a way to invoke a Java method from within a query. This creates an option to extend the relational algebra with procedural capabilities that can be both complex and impossible to realize using an *EQL* query.

A *User-Defined Function* is implemented as a method that belongs to a class that extends *UserDefinedFunction* from the package *com.softwareag.wep.resource.udf*. A simple example is shown in Listing 3.3. The method needs to be annotated with *UserDefinedFunction* to make it available for use from within queries. The name of the particular function is defined with the attribute *name*, as can be seen in line 8. In this instance, the name of the function is defined as *unitStepFunction* and it is therefore now available in *EQL* queries, invoked as *UDF_unitStepFunction*. The Java method can have a predefined number of parameters that may be one of the following data types: *Integer*, *Double*, *Float*, *Long*, *Boolean*, *Byte*, *Short*, *BigDecimal*, or *String*. The return value can only be of the same data types. Due to the data type restriction, this method – which can handle any complex data type internally – is limited in communicating with a query. *UDF*s can only return a single value to a query, unlike the pattern matching, which can return as many variables as are defined in the *MEASURES* clause. This greatly reduces the possible use cases for this method unless alterations are made in its implementation.

```
1 package com.softwareag.UDFExample;
2
3 import com.softwareag.wep.resource.udf.UserDefinedFunction;
4 import com.softwareag.wep.resource.udf.UserDefinedFunctions;
5
6 public class UDFContainer extends UserDefinedFunctions {
```

```
7
8   @UserDefinedFunction(name = ‘‘unitStepFunction’’)
9   public static Integer unitStepFunction(Double value) {
10    if (value >= 0.0)
11      return 1;
12    else
13      return 0;
14  }
15 }
```

Listing 3.3: Example of a *User-Defined Function* implementation.

Once the query is implemented, as shown in Listing 3.3 on page 42, it can be called from within an *EQL* query. Listing 3.4 shows a simple example of how to use a *UDF* in an *EQL* query. In line 6, the unit step function is called, and the return value is directly used in a conditional clause. *User-Defined Functions* can also be called from within pattern matching and *SELECT* clauses. Multiple parameters in a *UDF* are comma separated.

```
1 SELECT
2   *
3 FROM
4   Bid
5 WHERE
6   UDF_unitStepFunction( Bid_Price - 200.0 ) = 1;
```

Listing 3.4: Example of a query utilizing a *User-Defined Function*.

Use of *UDF*s enables complex computations to be moved and handled with a Java method. While this Java method is able to access whole frameworks and even use static variables for storage of values, both are discouraged. The *UDF* is intended to return deterministic and reproducible results, as a function in the mathematical sense. Therefore, the results should be solely dependent on the parameters that are passed to the *UDF*.

All but one of the presented algorithms return a single best value after receiving a new piece of data and because of this, can be easily implemented using *UDF*s. Weights, full *Decision Trees*, and a list of previously entered data can be stored in Java and used for any future calculation. If the calculation is not of linear complexity, or even if the amount of data in a single execution of an algorithm becomes very large, the *UDF* can slow the execution of the query, leading to temporal inconsistencies. Offline algorithms would require the storage of all data in the *UDF*. Although this is possible, it is greatly discouraged and should not be used. For offline algorithms, the data for one execution tends to be rather large, but also consists of data collected during a longer time frame. Online

algorithms, on the other hand, can become more complex when viewed as *calculations required per single piece of data*. They can also be seen as calculations based on the item set generated by a window of one element (*WINDOW (1 ROW)*).

3.7 CEP Queries Using User-Defined Aggregate Functions

In Martin Prodanov's thesis [Prodanov, 2011], he also describes the *User-Defined Aggregate Function*s; however, not much documentation is yet available. Aggregate function, e.g., *AVG*; *MIN*; and *MAX*, can capture multiple input elements and calculate an aggregated result. Like *UDF*s, *UDA*s are written in Java. An interface must be implemented that consists of two methods: *aggregate* and *getAggregate*. The first method is always called when an event arrives and, simultaneously, is also handed an array of currently valid inputs. The *CEP* engine handles the validity of events and thus is not called to handle this issue. Whenever a value is required in a *CEP* query for output, the method *getAggregate* is invoked. The data type restrictions are the same as for *UDF*s.

*UDA*s complement *UDF*s with the capability to handle multiple events at the same time, making them more powerful. Also, the fact that the engine always sends a full set of valid elements eases the managing work required by the Java programmer. Since a cluster of data always arrives at the method whenever a new event arrives, the algorithm used here should be an offline algorithm that works with linear complexity. In order to use an online algorithm, the class must keep track of the elements that are becoming invalid as well as the new one arriving, which adds managing complexity that is better handled in *UDF*s. Since the *CEP* engine handles the validity, the calculated results are much more dynamic in nature and better reflect the current environment than a general environment.

3.8 CEP Queries Using User-Defined Operators

The *User-Defined Operator* (*UDO*) is a construct currently in the final stages of development. It is a part of the *EQL* syntax that allows for more complex computations on data sets. A *User-Defined Operator* can handle a greater number of events simultaneously, as well as perform several computations on the given set before creating an output. This output can also be a set of one or more events of the output type. This means the *UDO* is the only extension of *Software AG's CEP* that can produce multiple events. Documentation on *UDO*s is still sparse due to its developmental stage: this description is based on the Master's Thesis by Martin Prodanov [Prodanov, 2011].

The *UDO* is both the most complex and most powerful tool in *CEP*. However, by using *UDO*s, the system can lose its snapshot reducibility, because the returned events can have a later timestamp due to the computation and delayed return of results. If the resulting complex events are not put into direct relation to the incoming events, the side effects, if

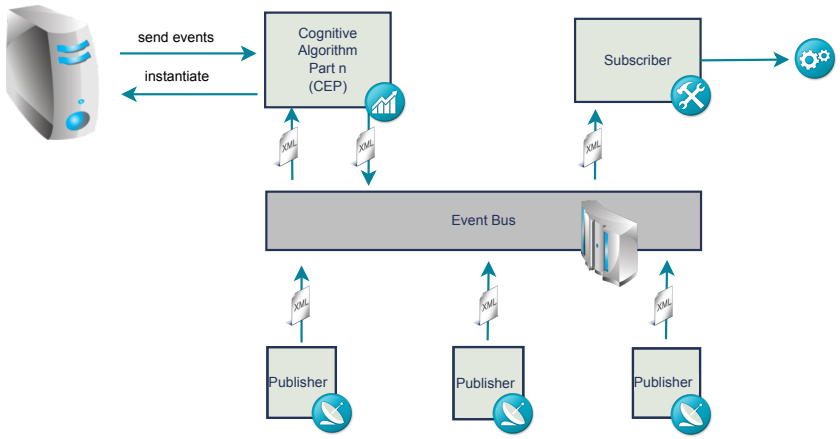

Figure 3.8: Realization of a cognitive algorithm in *CEP* using *UDO*s.

any, are negligible. Otherwise this results in a violation of the rule of consistency, hence the loss of snapshot reducibility.

The structure of a *UDO*, shown in Figure 3.7 on page 42, is very similar to a *UDF*; however, it returns an event stream to a *CEP* query instead of a single value. It actually receives every incoming event after its instantiation and thereafter sends events back to the query as shown in Figure 3.8. The exact definition of a *UDO* is done in a Java class that implements the *UserDefinedOperatorAdapter* shown in Listing 3.5, however, it is much more complex than a *User-Defined Function*.

A *UDO* receives incoming events through the *push* method. It receives them individually and only receives the event value and validity interval. It does not have to handle the conversion from *XML*, which a general subscriber would have to do. The push method does not automatically send events to the event bus, but leaves this decision to the implementation of the *UDO*. In order to return an event to the query, the same callback object is used that was handed to the *UDO* by the open method, and was already used by the *CEP* engine to initialize the *UDO*. The events are sent to the event bus in a synchronous way which prevents a data flood. Inside the *UDO*, events are represented as instances of the *Element* class.

```
1 public interface UserDefinedOperatorAdapter {
2   int getMinimumRequiredNumberOfInputs();
3   int getMaximumAllowedNumberOfInputs();
4   StreamMetaData computeOutputSchemaFromInputSchemas
5     (StreamMetaData... inputStreamMetadata) throws Exception;
6   Object getInputType(int index);
7   void init();
8   void open(Callback callback);
```

```
 9   void push(int index, Element<?> element);
10   void heartBeat(int index, long timeStamp);
11   void done(int index);
12   void close();
13 }
```

Listing 3.5: Interface to implement UDOs.

To solve the problem of a missing sort implementation in *CEP*, a generic *UDO* implementation – the *topkanalyzer* – determines the best k elements in a stream.

On the one hand, a *UDO* does have the advantage that complex event processing has. On the other hand, the use of overly complex algorithms, or anything capable of creating non-deterministic results, is highly discouraged.

3.9 Event Processing Using a Custom Component Utilizing JMS Adapters

Another option for event processing is custom design and implementation of *Complex Event Processing* outside the actual *CEP* engine. A custom component communicates with the event bus using hard-coded *JMS* subscribing and publishing adapters. The configuration of the adapters is dependent on the event bus installation. Figure 3.9 on page 47 shows how this component is placed in the *CEP* environment.

A subscriber is configured to receive all events to a given topic. After receiving the topic, it is sent to the core of the component, where the calculations take place. The calculated output is re-entered onto the event bus using the publisher.

Such a custom component may be used for various reasons. Figure 3.6 on page 39 demonstrated a partial implementation of the component, which stored received events directly into a database. A following *CEP* query then utilized the database entries.

The unique advantage of this component is the freedom it offers its implementer. Everything can be custom made, and the use of external frameworks is easily realizable. The custom component can be used as the missing link between the *Apache Mahout* framework and the *CEP*, combining the static world of a framework with the dynamic *CEP*.

Some of the disadvantages of using this method are the high costs for the realization of individual tasks; the loss of the advantages of using standardized *CEP*; and the possible loss of snapshot reducibility and consistency. However, snapshot reducibility can also be lost with the usage of any *CEP* function.

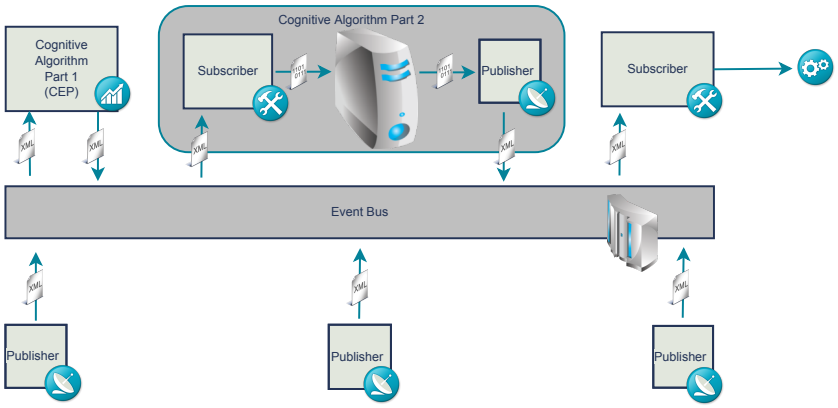

Figure 3.9: A *CEP* using a custom component utilizing JMS adapters.

3.10 Proposal for the Realization of the Presented Algorithms in CEP

This section describes a two-step method for determining which functions are best suited for the realization of a new algorithm. The method is based on the knowledge gained about each function.

1. Split the Algorithm into its Components

First, the algorithm is split into simple components, with each component represented by an input, a computation, and an output. The cardinality for input and output is important, as is their relationship to each other.

2. Find the Best Angle for every Component

In the second step, the components are analyzed individually using Table 3.1 on page 48 to determine the right method. In this table, *Custom* represents a customer processing unit that utilizes *JMS* adapters, as shown in section 3.9. Often it is beneficial to eliminate any method marked with two minuses, but in some cases a trade-off is necessary to find the right method.

Given the aforementioned conceptual options available under *Software AG's CEP*, the following solutions are proposed for usage with the shown cognitive algorithms.

The *k-Nearest Neighbor* calculates the *Euclidian* distance to all available, previously classified elements. It is an algorithm that, once trained, receives a new event, calculates the best fitting class, and returns that class. This involves calculating the distance to every classified element, sorting those results, and determining the most frequent class among the closest k elements. The *Euclidian* distance between two elements is a typical example for

	CEP	DB	UDF	UDA	UDO	Custom
Can the algorithm be realized as a SQL query without sorting?	++	+	○	– –	○	– –
Does the algorithm need some type of sorting to determine the best elements?	–	– –	++	++	++	++
Can one input lead to many outputs?	– –	– –	– –	– –	++	+
Are complex algorithms or data structures required?	– –	– –	++	++	++	+
Is the system moved or copied frequently	++	++	++	++	++	– –
Does the algorithm use weights or other static variables that need frequent updates?	-	+	++	○	+	++
Shall the implementation be comprehensive and maintainable?	++	++	+	+	○	– –
Does the algorithm use a large set of master data?	– –	++	○	○	○	– –

Table 3.1: Classification of *CEP*s features by properties of cognitive algorithms.

a *UDF* calculation. The closest k elements can be found using the *UDO* implementation *topkanalyzer*. To store existing data with known classifications, a custom component can load the events and store their data in a database.

The *Decision Trees* require a cluster of training data before they are capable of building reliable *Decision Trees*. This can best be achieved in a custom component, where the calculation of the entropy is also easily realized. The calculation of the tree is rather complex, and should only be executed once, perhaps with an initializing event. The categorization of newly incoming events is also easily achieved, and only a single value – the best fitting category – is returned to the query. Once trained, a *UDF* is able to execute the decision making, based on the predetermined *Decision Tree*. In that case, the transfer of the tree is a negligible use of external data and will still produce deterministic results.

The *Naïve Bayes* algorithm is also best realized in a custom component. In Harrington's example, the *Naïve Bayes* was used to classify text from blogs as appropriate and inappropriate. The simplest approach in this case would be to send training data into a custom component containing strings of the blog entries, possibly classifying them with numbers representing statuses.

In *Logistic Regression* the sigmoid function and the weight storing is best performed in *UDF* during the training part of the algorithm. A custom component offers the best results for training the algorithm. Once the training is complete, the weights can be transferred into a database for quicker access. From the database, they can be used directly within a *CEP* query that uses either the step function from *UDF* or a simple version of a step function directly inside the query.

The *Support Vector Machines* are also best realized using custom components. The complex calculations with matrices, and storing the weights for the hyperplanes are best accomplished in Java: Java classes can facilitate both.

As with *Logistic Regression*, the suggested approach to *Linear Regression* is to perform the training in custom components, and store the calculated weights into a database. From this database, a *CEP* query is able to calculate results using incoming events without further use of the custom component.

The *k-Means Clustering* is an unsupervised algorithm that is continuously able to update its cluster means. The calculation of a mean can be performed in *CEP* and the results for the means of clusters can be stored in a database using a custom component. This approach, using tumbling windows, can help create very dynamic, i.e., dynamically updating, means for the clusters. On the other hand, the mean values of certain clusters could eventually migrate to extremes of the scale, rendering them useless. The *Bisecting k-Means Clustering* must be carried out in a custom component, since, until k clusters

Algorithm	Proposed Solution
k-Nearest Neighbor	UDF, custom component, UDO, and database
Decision Trees	Custom component UDF possible in productive use
Naïve Bayes	Custom component
Logistic Regression	Custom component and database in training CEP, database, and UDF in productive use
Support Vector Machines	Custom components
Linear Regression	Custom components and database in training CEP and database in productive use
k-Means Clustering	CEP, custom component, and database or the entire algorithm in a custom component
Apriori Association Analysis	Custom component or UDO

Table 3.2: Proposed usage of presented algorithms in *Software AG's CEP*.

exist, the sequential steps for splitting clusters cannot be achieved with relational algebra in a continuous event flow.

The calculations performed in an *Association Analysis* are best handled in Java. In this instance, the best choice is the complex custom component, as compared to a *UDO* implementation, due to multi row results. The *Association Analysis* is most attractive for hourly, daily, or longer time intervals.

Custom components present good possibilities for realizing cognitive algorithms in *CEP*. They allow the greatest freedom for implementation since, unlike *UDA*s, they are not restricted only to handling valid items. Additionally, they present a better option for the implementation with regard to sequencing computation. In some cases, the results calculated in custom components can be stored in a database to speed up future calculations in a *CEP*, rendering custom components obsolete in productive use. *UDO*s are also a good choice to realize cognition, especially when many outgoing complex events, produced by a single incoming event, are desired or anticipated.

The analysis of the algorithms above has referenced the general process for a cognitive algorithm using *CEP*, as shown in Figure 3.10 on page 51. It is worth noting that the figure is simplified for the purpose of illustration. In any given realization, depending on the scenario, some steps may have several additional components assigned to them, while others may not. In the aforementioned scenarios, the data originates from sensors and is provided to the algorithm via the event bus. Once inside the system, any component can read and write data to both the event bus and a database. First, either through aggregation or via monitoring, the data is reduced only to that which specifically relates

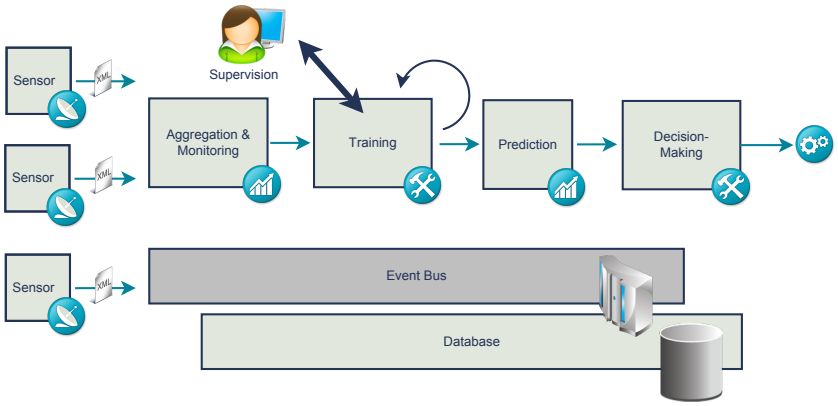

Figure 3.10: General implementation for a cognitive algorithm in *CEP*.

to the following steps. Training, based on the data, occurs in the second step. Frequently, the training is a repetitive process that runs the data numerous times. In a supervised training, a user sets the results for incoming input and controls the learning. In contrast, an unsupervised algorithm does not make use of inputs from users. In the third step, a prediction is made based on the input data. Finally, the decision component generates an action to be made based on the prediction.

3.11 Summary

Cognitive algorithms can be realized in various functions provided by *CEP*. This chapter analyzes the capabilities of each function, as required for the proposed example algorithms. Each function has specific advantages and disadvantages, which are discussed, and a methodology is proposed to assist in choosing the optimal function for a given problem.

4 REALIZATION

In order to demonstrate the realization of the proposed concepts to integrate cognitive technology into the *CEP*, two sample algorithms have been selected. The first algorithm is the *Slope One* classification algorithm, also used in the Apache Software Foundation's *Apache Mahout* project. It was presented in section 2.3.1. The second algorithm is a *Fire Detection System Based on Intelligent Data Fusion Technology* algorithm. It uses information from three different kinds of fire sensors and combines the data into its calculations to create a more precise fire detection system with fewer false alarms. This algorithm was presented in section 2.3.2.

First, the connection to the event bus, which is used to publish the events, is defined. This is achieved by using *Software AG*'s *JMS* implementation webMethods Broker.

In the second section, the realization of the *Slope One* algorithm is shown, using different approaches. The structure and access of the sample data is displayed first, followed by a description of the implementation in the *Apache Mahout* framework. Then the realization with a relational database is presented. This method was chosen because of the relationship between the *DBMS* and *CEP*, and because of the relational query language *EQL* used in *Software AG's CEP*. The database chosen is a *MySQL* database, because the dialect of *MySQL* is most similar to that of *EQL*. The next subsection demonstrates the realization of the *Slope One* algorithm in *Software AG's CEP* by transforming and reusing the queries that were used in *MySQL*. An example of a database connection that is used to process and integrate master data into the complex events follows.

Last, the *Fire Detection System Based on Intelligent Data Fusion Technology* algorithm by Bao et al. demonstrates the use of pattern matching and *User-Defined Function*s in *CEP*.

4.1 JMS Adapter to Publish and Subscribe Events on the Event Bus

To communicate events to and from an event bus, an adapter needs to be provided, both for the publisher and the subscriber. Both adapters are realized using *Software AG*'s implementation of *JMS* – the *webMethods Broker*. This Broker can be used to send and receive any data through the event bus on a topic to which it is registered. In order for the

data to become an event that can be interpreted and used by a *CEP* engine, it must be converted to a string containing an *XML* file. The *XML* content of a string has to follow the structure of the envelope and the schema definition of the event type chosen. The information is split into a header – containing the meta information about the event – and the body – containing the actual content of the event.

The higher level structure of the envelope is defined in the *XML Schema Definition* "Envelope.xsd". The structure of the envelope is shown in Figure 4.1 on page 54, and the full document is shown in the appendix in section 5.5. It defines how the meta data of the event is structured, containing information about the start and end time, the type of event, and the version of the definition. This is how the header of any event is structured.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
3    xmlns:eda="http://namespaces.softwareag.com/EDA/Event"
4    xmlns:p="http://namespaces.softwareag.com/EDA"
5    targetNamespace="http://namespaces.softwareag.com/EDA"
       elementFormDefault=‘‘qualified’’>
6    <xs:import namespace="http://namespaces.softwareag.com/EDA/Event
       " schemaLocation="Event/Envelope.xsd"/>
7    <xs:element name=‘‘Ratings’’ type="p:RatingsType"
       substitutionGroup="eda:Payload"/>
8    <xs:complexType name=‘‘RatingsType’’>
9      <xs:sequence>
10       <xs:element name=‘‘UserID’’ type="xs:long"/>
11       <xs:element name=‘‘MovieID’’ type="xs:long"/>
12       <xs:element name=‘‘Rating’’ type="xs:float"/>
13       <xs:element name=‘‘Tstamp’’ type="xs:long"/>
14     </xs:sequence>
15   </xs:complexType>
16 </xs:schema>
```

Listing 4.1: Event type definition for *Ratings* events.

The body of an event is defined in individual schema definitions. Listing 4.1 shows the definition of the event type *Ratings* in the form of an *XML* schema definition. This event type is later used in the *Slope One* example. The name of the event type is located in the main attribute of the highest level element: *xs:element*. The file must be named the same way as the event type – here *Ratings.xsd*. The structure of the event is described in the element *xs:complexType*, which carries the attribute *name* containing the name of the event type, plus "Type", here *name=RatingsType*. The structure below contains two different kinds of elements – fields and composites. Both elements have the same *XML*

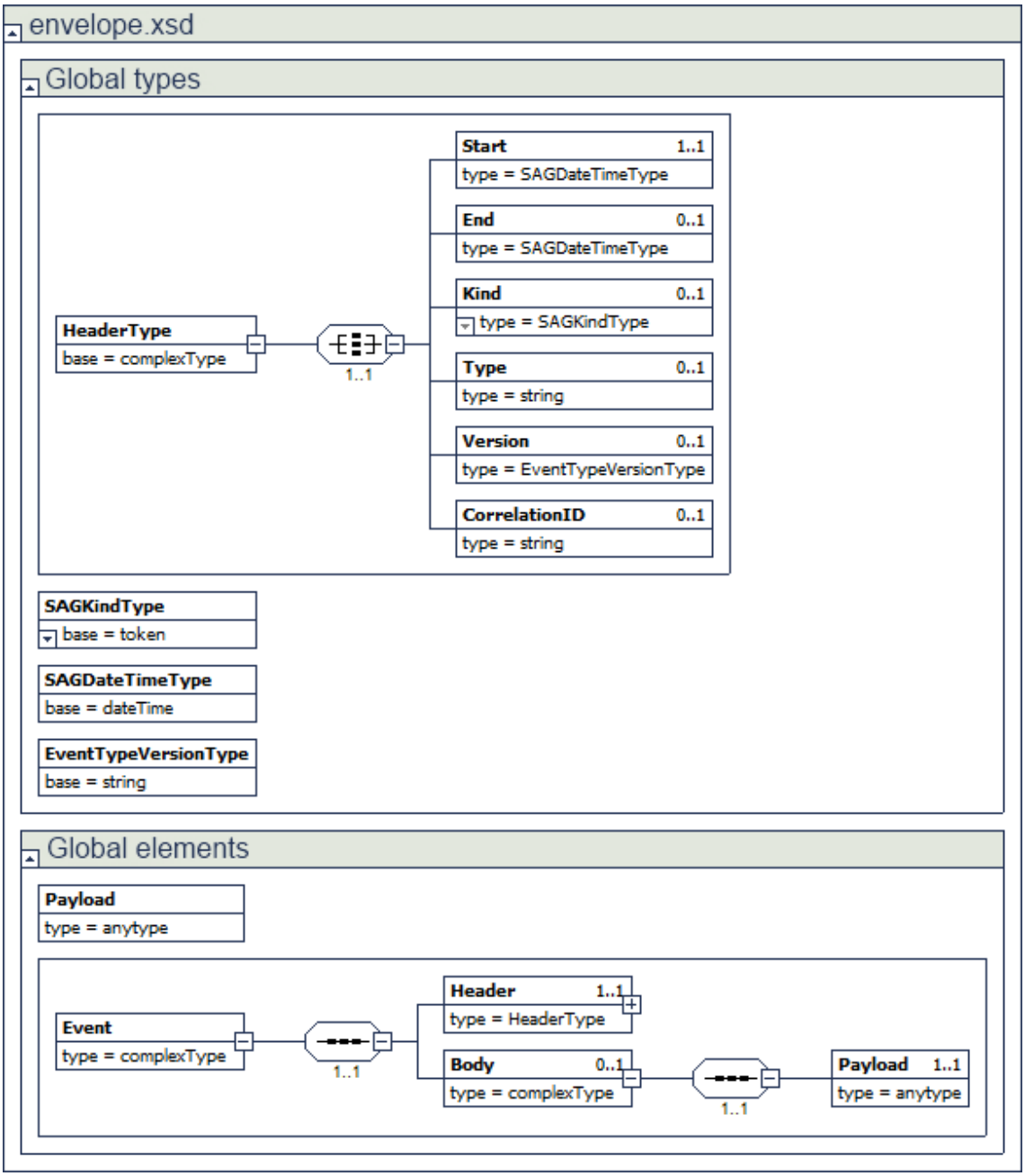

Figure 4.1: Structure of envelope.xsd.

element *xs:element*, but only the field has an attribute *type* to define the data type used for the element. Composites have a child element, *xs:complexType*, that again contains fields and composites. Although this could be used to describe a tree structure, here it is internally interpreted as a sequential list.

Listing 4.2 displays a sample event that follows the structure defined in Listing 4.1 on page 53. It shows an event of the event type *Ratings*, which was provided with a timestamp set to a time in the year 1997, and lasts for a millisecond, due to the lack of a defined end time in the header.

```
1  <?xml version="1.0" encoding="UTF-8" standalone=‘‘yes’’?>
2  <eda:Event xmlns:eda="http://namespaces.softwareag.com/EDA/Event"
3    xmlns="http://namespaces.softwareag.com/EDA">
4    <eda:Header>
5      <eda:Start>1997-09-20T03:05:10.000Z</eda:Start>
6      <eda:Kind>Event</eda:Kind>
7      <eda:Type>Ratings</eda:Type>
8      <eda:Version></eda:Version>
9      <eda:CorrelationID></eda:CorrelationID>
10   </eda:Header>
11   <eda:Body>
12     <Ratings>
13       <UserID>259</UserID>
14       <MovieID>255</MovieID>
15       <Rating>4.0</Rating>
16       <Tstamp>874724710</Tstamp>
17     </Ratings>
18   </eda:Body>
19 </eda:Event>
```

Listing 4.2: Example event as processed inside the *CEP*.

Listing 4.3 on page 56 shows a *JMS* adapter that both publishes and receives texts on the topic *Event::Ratings*. The topic is part of the event bus group *EventMembers* on the event bus *Broker #1*, which runs on the same host on port *6849*. This is defined in the static variables in lines 8 through 11 before the connection is configured, and established in the method *startConnection* beginning in line 17. Text messages are published using the method *publishTextMessage* in line 39, then received and printed with method *enableConsumer* in line 48.

```
1  import javax.jms.*;
2
3  import com.webmethods.jms.WmTopicConnectionFactory;
4  import com.webmethods.jms.impl.WmTopicConnectionFactoryImpl;
5
6  public class JMSClient {
7
8    private static String eventBusName = "Broker #1";
9    private static String eventBusHost = "localhost:6849";
10   private static String eventBusGroup = ‘‘EventMembers’’;
11   private static String eventBusTopic = "Event::Ratings";
12
13   private Connection connection;
14   private Session session;
15   private Topic topic;
16
17   public void startConnection( String eventBusName , String
       eventBusHost , String eventBusGroup , String eventBusTopic )
       {
18     if ( connection != null ) return;
19     try {
20       WmTopicConnectionFactory wmTopicConnectionFactory = new
           WmTopicConnectionFactoryImpl();
21       wmTopicConnectionFactory.setBrokerName( eventBusName );
22       wmTopicConnectionFactory.setBrokerHost( eventBusHost );
23       wmTopicConnectionFactory.setClientGroup( eventBusGroup );
24       connection = wmTopicConnectionFactory.createConnection();
25       connection.start();
26       System.out.println( "Connected to " + eventBusName + " on " +
             eventBusHost );
27       session = connection.createSession( false , Session.
           AUTO_ACKNOWLEDGE );
28       topic = session.createTopic(eventBusTopic);
29     } catch ( JMSException e ) { e.printStackTrace(); }
30   }
31
32   public void stopSessionAndConnection() {
33     try {
34       if ( session != null ) session.close();
35       if ( connection != null ) connection.close();
36     } catch ( JMSException e ) { e.printStackTrace(); }
37   }
```

```
38
39  public void publishTextMessage( String messageString ) {
40    try {
41      MessageProducer messageProducer = session.createProducer(
          topic );
42      Message message = session.createTextMessage( messageString );
43      messageProducer.send( message );
44    }
45    catch ( JMSException e ) { e.printStackTrace(); }
46  }
47
48  public void enableConsumer() {
49    try {
50      MessageConsumer messageConsumer = session.createConsumer(
          topic );
51      messageConsumer.setMessageListener( new MessageListener() {
52        public void onMessage( Message message ) {
53          try {
54            if ( message instanceof TextMessage )
55              System.out.println( ( ( TextMessage ) message ).
                  getText() );
56            else
57              System.out.println( message.getJMSMessageID() );
58          } catch ( JMSException e ) { e.printStackTrace(); }
59        }
60      });
61    }
62    catch ( JMSException e ) { e.printStackTrace(); }
63  }
64
65  public static void main( String[] args ) {
66    try {
67      JMSClient jmsClient = new JMSClient();
68      jmsClient.enableConsumer();
69      for ( int i = 0 ; i < 50 ; i++ ) {
70        jmsClient.publishTextMessage( "message #" + i );
71        Thread.currentThread().sleep( 1000 );
72      }
73    } catch ( InterruptedException e ) { e.printStackTrace(); }
74  }
```

```
75 }
```

Listing 4.3: Simple connection program to demonstrate a *JMS* connection to the event bus.

In order to send well-formed events which can be processed in the *CEP* engine onto the event bus, the events need to be packaged according to the structure defined in *Envelope.xsd*. This is outlined in Listing 4.4 and is a generic packager that is independent of the event's own structure. The incoming parameter, *eventBody*, expects a perfectly formed and packaged event body. This method does not define an end timestamp. If an end timestamp is not defined, the *CEP* engine assumes the minimal time interval for the event, which is 1 ms. Line 16 shows how to insert an optional end timestamp.

```
1  protected static String packageEvent( String eventBody , long
       start , String schemaNamespace , String schemaLocation ) {
2    SimpleDateFormat dateFormat = new SimpleDateFormat( "yyyy-MM-dd'
       T'HH:mm:ss.SSS'Z'" );
3    dateFormat.setTimeZone( TimeZone.getTimeZone( ``GMT'' ) );
4    StringBuffer xmlEvent = new StringBuffer();
5    xmlEvent.append( "<?xml version=\"1.0\" encoding=\"UTF-8\"
       standalone=\"yes\"?>\n" );
6    xmlEvent.append( "<eda:Event xmlns:eda=\"http://namespaces.
       softwareag.com/EDA/Event\"\n" );
7    xmlEvent.append( "       xmlns=\``' + schemaNamespace + "\``'
       );
8    xmlEvent.append( ">\n" );
9
10   // The header of the event
11
12   xmlEvent.append( " <eda:Header>\n" );
13   xmlEvent.append( "  <eda:Start>" );
14   xmlEvent.append( dateFormat.format( start ) );
15   xmlEvent.append( "</eda:Start>\n" );
16   // xmlEvent.append( " <eda:End>" + dateFormat.format( end ) +
       "</eda:End>\n" );
17   xmlEvent.append( "  <eda:Kind>Event</eda:Kind>\n" );
18   xmlEvent.append( "  <eda:Type>Ratings</eda:Type>\n" );
19   xmlEvent.append( "  <eda:Version></eda:Version>\n" );
20   xmlEvent.append( "  <eda:CorrelationID></eda:CorrelationID>\n" );
21   xmlEvent.append( " </eda:Header>\n" );
22   xmlEvent.append( " <eda:Body>\n" );
23
24   // The body of the event
```

```
25
26    xmlEvent.append( eventBody + "\n" );
27    xmlEvent.append( " </eda:Body>\n" );
28    xmlEvent.append( "</eda:Event>" );
29    return xmlEvent.toString();
30  }
```

Listing 4.4: The generic event packager method.

4.2 Slope One Classification Algorithm

This section demonstrates how the *Slope One* algorithm is realized through various means. First the structure of the data is explained, as well as the method by which it was transferred into the structure in the format required for further use. To ensure maximum comparability, the same data and its structure is applied that was used in the *Apache Mahout* example implementation. Then the realization of the algorithm in its original *Apache Mahout* implementation is briefly discussed. A realization with a *MySQL* database precedes the description of how *CEP* queries can be used to realize the algorithm. Finally, the *CEP* query is shown with an integrated database that allows access to the master data for the incoming events.

4.2.1 The Sample Data

The recommender of movies is used as an example in the *Apache Mahout* project for recommending movies to users of a movie rating website. Based on previous ratings, and compared to other users who have rated the same movies, a user receives predictions of movies that he or she might enjoy. The data comes from the movie ratings webpage, movielens, which is a free service by the Grouplens Research at the University of Minnesota and is published at `http://www.grouplens.org/node/73`. The data is split into three data sets with varying sizes of 100k ratings, 1M ratings, and 10M ratings. The structure of the data varies to some degree between the data sets and the 1M ratings structure used in this example.

The data consists of user data, movie data, and movie ratings by users. All files are sequential files with one tuple per line. The user and movie data is mainly used for other algorithms that involve either item or user similarity.

Users: **UserID::Gender::Age::Occupation::Zip-code**

The user data reflects basic information that can be utilized to cluster users while still maintaining anonymity. A *UserID* is used to identify the user. Gender is denoted with *M*

for male and *F* for female users. The age of the user is classified into seven groups which are represented with a predefined integer number within their respective age ranges. The occupation portion contains a code for 21 different predefined occupations. Last, the Zip-Code identifies user groups based on their location in the U.S. and contains both five-digit ZIPs, and nine-digit ZIP+4s.

Movies: **MovieID::Title::Genres**

Movies only contain their ID, full title and a list of genres. The genres are described by 18 predefined strings, and a movie can belong to one or more genres.

Ratings: **UserID::MovieID::Rating::Timestamp**

The Ratings contain both the ID of the user and the movie, plus the timestamp for identification of individual ratings. The timestamp is the number of seconds since the start of the Unix epoch (1970-01-01 UTC) as is returned by the Linux command program *date +%s*. The rating itself represents the rating the user has selected and is an integer representing a five star rating system assigning whole stars only.

4.2.2 Original Apache Mahout Implementation

The *Slope One* algorithm in *Apache Mahout* is implemented in the typical manner as seen in Figure 3.1 on page 28. All the data is read into the memory using the interface *DataModel* in the package *org.apache.mahout.cf.taste.model*. An excerpt of this data model interface, which is licensed under the Apache License 2.0 to the Apache Software Foundation, is shown in Listing 4.5. The data structure is highly specific to the algorithm, and so has been optimized for both speed and memory usage. It allows a quick read in, and parallel primary calculation of the deviation between the items as described in equation 2.1. This represents the training set, and for every updated or new rating it can also be dynamically updated. Thus, the implementation of this algorithm works offline as well as online.

```
1 package org.apache.mahout.cf.taste.model;
2
3 import java.io.Serializable;
4
5 import org.apache.mahout.cf.taste.common.Refreshable;
6 import org.apache.mahout.cf.taste.common.TasteException;
7 import org.apache.mahout.cf.taste.impl.common.FastIDSet;
8 import org.apache.mahout.cf.taste.impl.common.
     LongPrimitiveIterator;
9
```

```
10 public interface DataModel extends Refreshable, Serializable {
11
12   LongPrimitiveIterator getUserIDs() throws TasteException;
13
14   PreferenceArray getPreferencesFromUser(long userID) throws
        TasteException;
15
16   FastIDSet getItemIDsFromUser(long userID) throws TasteException;
17
18   Long PrimitiveIterator getItemIDs() throws TasteException;
19
20   PreferenceArray getPreferencesForItem(long itemID) throws
        TasteException;
21
22   Float getPreferenceValue(long userID, long itemID) throws
        TasteException;
23
24   Long getPreferenceTime(long userID, long itemID) throws
        TasteException;
25
26   int getNumItems() throws TasteException;
27
28   int getNumUsers() throws TasteException;
29
30   int getNumUsersWithPreferenceFor(long itemID) throws
        TasteException;
31
32   int getNumUsersWithPreferenceFor(long itemID1, long itemID2)
        throws TasteException;
33
34   void setPreference(long userID, long itemID, float value) throws
         TasteException;
35
36   void removePreference(long userID, long itemID) throws
        TasteException;
37
38   boolean hasPreferenceValues();
39
40   float getMaxPreference();
41
42   float getMinPreference();
```

```
43 }
```

Listing 4.5: Excerpt from the Apache Mahout project data model for taste recommenders.

To receive a prediction, a request must be sent to *Apache Mahout* for a given user utilizing the method *public List<RecommendedItem> recommend(long userID, int howMany, IDRescorer rescorer) throws TasteException* in the main class *SlopeOneRecommender* in the package *org.apache.mahout.cf.taste.impl.recommender.slopeone*. The requester specifies the user for whom a recommendation is to be supplied and the number of desired recommendations, thus actively making an information request to the cognitive algorithm.

4.2.3 Database Implementation

The *Slope One* algorithm used within *Apache Mahout*'s recommender can be reduced to two formulas as seen in the publication by Lemire and MacLachlan [Lemire and MacLachlan, 2005]. This is shown in section 2.3.1. These two formulas can be translated into an *SQL* query, and executed by a regular relational DBMS. The following description shows how this algorithm functions in a *MySQL* environment. *MySQL* has been chosen because its syntax is similar to *EQL*'s syntax.

As a first step, a table must be created, in which the ratings are stored and from which the queries are later executed. The structure was described in section 4.2.1 but remains to be implemented in *SQL*. Since movie ratings are the sole interest for the *Slope One* algorithm, only the *Ratings* table shown in Listing 4.6 is used.

```
1  -- The table for the ratings.
2  CREATE TABLE Ratings
3  (
4    UserID bigint NOT NULL,
5    MovieID bigint NOT NULL,
6    Rating double precision,
7    Timestamp bigint,
8    CONSTRAINT RatingPK PRIMARY KEY (UserID , MovieID),
9    CONSTRAINT RatingMovieFK FOREIGN KEY (MovieID)
10     REFERENCES Movies (MovieID) MATCH SIMPLE
11     ON UPDATE NO ACTION ON DELETE NO ACTION,
12   CONSTRAINT RatingUserFK FOREIGN KEY (UserID)
13     REFERENCES Users (UserID) MATCH SIMPLE
14     ON UPDATE NO ACTION ON DELETE NO ACTION
15 );
```

Listing 4.6: *MySQL* query to create a table for ratings.

In order to insert the data from the file provided by the Grouplens Project, a Bash command line script has been written to translate the sequential data file into an insert script for *MySQL* as shown in Listing 4.7 on page 63. The original file was *ratings.dat* and each line is translated into an individual insert statement using a regular expression.

```
perl -pe "s/^(.*)::(.*)::(.*)::(.*)$/\
INSERT INTO Ratings \
(UserID, MovieID, Rating ,Timestamp) \
VALUES (\1, \2, \3, \4);/g" ratings.dat > Ratings.sql
```

Listing 4.7: Perl command line script to create *MySQL* insert script for the ratings.

Once the table is created, and the data is read in, the algorithm can be applied. The first step calculates the item deviation as shown in equation 2.1 on page 20, followed by the second step in which the prediction is calculated. The item deviation computation is complex and results in a calculation with a square complexity in an SQL query, $O(n^2)$, as seen in Listing 4.8. The query takes every combination of two movies rated by the same user and averages the difference in ratings between them. To reduce the amount of time required for the calculation, the query has been limited to the ratings of the first twenty users, sorted by their ID. This is when the optimized calculation of the deviation of the *Apache Mahout* algorithm during load time becomes advantageous.

```
SELECT
  r1.MovieID AS Item1,
  r2.MovieID AS Item2,
  AVG(r1.Rating - r2.Rating) AS ItemDeviation
FROM
  Ratings AS r1,
  Ratings AS r2
WHERE
  r1.UserID < 21 AND
  r2.UserID < 21 AND
  r1.UserID = r2.UserID AND
  r1.MovieID != r2.MovieID
GROUP BY
  Item1,
  Item2;
```

Listing 4.8: *SQL* query to produce a general preference deviation between items.

To use the item deviation in the upcoming algorithm, which will then calculate the predictions, the result of the query in Listing 4.8 needs to be provided to the next query. This

could be achieved in three ways. The result may be directly queried using a view; the query may be run periodically; or the query may be triggered by incoming ratings with the results stored in a separate table. The first two approaches carry a large number of calculations into the query for the prediction as required by the square complexity of the item deviation. This prolongs the query unnecessarily by the sheer volume of calculations. Furthermore, the item deviation calculation is the more complex of the two queries. Additionally, the item deviation would represent the common element in every prediction query, creating an overhead of multiple redundant calculations of the same algorithm. Therefore, the results should be stored in a separate table, as defined in Listing 4.9 on page 64. Triggered queries enable the *DBMS* to react to incoming events similar to a *CEP*. However, the frequency of incoming events in an *IoT* environment would exceed the capabilities of classic *DBMS*s.

```
1  -- Creation of the ItemDeviation table.
2  CREATE TABLE ItemDeviation
3  (
4    Item1 bigint NOT NULL,
5    Item2 bigint NOT NULL,
6    ItemDeviation double precision,
7    CONSTRAINT ItemDeviationPK PRIMARY KEY (Item1 , Item2 ),
8    CONSTRAINT ItemDeviationFK1 FOREIGN KEY (Item1)
9      REFERENCES Movies (MovieID) MATCH SIMPLE
10     ON UPDATE NO ACTION ON DELETE NO ACTION,
11   CONSTRAINT ItemDeviationFK2 FOREIGN KEY (Item2)
12     REFERENCES Movies (MovieID) MATCH SIMPLE
13     ON UPDATE NO ACTION ON DELETE NO ACTION
14 );
```

Listing 4.9: Creation statement for the item deviation table in *MySQL*.

Using the query from Listing 4.8 on page 63 and adding a *REPLACE INTO ItemDeviation* before the actual query fills the new table with the item deviation values.

The publication first describes the general prediction algorithm in equation 2.2, before presenting the *Slope One* algorithm in equation 2.3. While the general prediction compares all items a user has rated with all those the user has not rated, the *Slope One* algorithm looks at all items the user hasn't rated, and adds the average item deviation to the user's average rating. Listing 4.10 on page 65 shows the first method; the general prediction algorithm. Two parameters must be hard coded into the query; one for the user that requires the predictions, and one for the number of required results. The user is defined in lines 9 and 12. In these two lines, subqueries create both the list of items already rated by that particular user, and the list of those not rated by that user. The number of desired results is defined by the *LIMIT* statement in line 21. It is important to note that the

ORDER BY statement in line 19 is the important feature distinguishing it from *EQL*. This enables it to receive an exact result of the algorithm directly without further sorting.

```
1  SELECT
2    id.Item1 AS Item1,
3    AVG(id.ItemDeviation + r.Rating) AS Prediction
4  FROM
5    ItemDeviation AS id,
6    Ratings AS r
7  WHERE
8    id.Item1 NOT IN
9      (SELECT DISTINCT MovieID FROM Ratings WHERE UserID = 1)
10   AND
11   id.Item2 IN
12     (SELECT DISTINCT MovieID FROM Ratings WHERE UserID = 1)
13   AND
14   r.UserID = 1
15   AND
16   id.Item2 = r.MovieID
17 GROUP BY
18   Item1
19 ORDER BY
20   Prediction DESC
21 LIMIT 10;
```

Listing 4.10: *MySQL* query to produce a preference prediction for user 1.

The query for the *Slope One* prediction is shown in Listing 4.11. As in the general prediction query, the *Slope One* prediction query has hard-coded both the values for the number of required results and the *UserID*. However, the complexity and the number of computations is reduced because only one sweep through the *ItemDeviation* table is now required.

```
1  SELECT
2    id.Item1 AS Item1,
3    (SELECT AVG(Rating) FROM Ratings WHERE UserID = 1) + AVG(id.
       ItemDeviation) AS Prediction
4  FROM
5    ItemDeviation AS id
6  WHERE
7    id.Item1 NOT IN
```

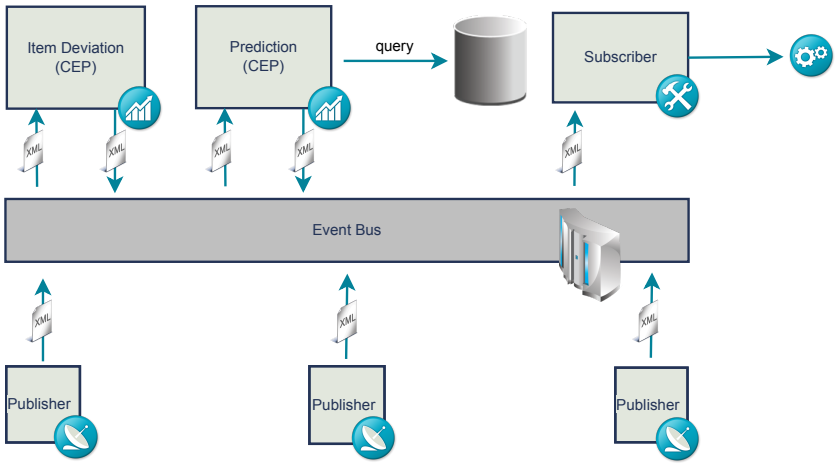

Figure 4.2: Realization of the *Slope One* algorithm with *CEP*.

```
8     (SELECT DISTINCT MovieID FROM Ratings WHERE UserID = 1)
9 GROUP BY
10   Item1
11 ORDER BY
12   Prediction DESC
13 LIMIT 10;
```

Listing 4.11: *MySQL* query to produce a *Slope One* prediction for user 1.

4.2.4 CEP Queries Realization

The implementation of the *Slope One* algorithm using *CEP* queries is, in many ways, similar to that of the database implementation. The structure of the data needs to be declared as well as the queries. Figure 4.2 displays the structure of the realization which already contains the database element for master data. Listing 4.1 on page 53 already contains the structure of the ratings event type in its *XML* form, and Listing 4.2 on page 55 shows an example event of that type. Compared to the *MySQL* table definition shown in Listing 4.6, an event for *CEP* does not have primary IDs, and the relation to other event types is not explicitly defined. An event type is simply the definition based on the order and data types of attributes in an event. The integration server running the event bus needs to be configured to expect events in an input stream. A topic is assigned to this input stream as well as the event type for the events, which are communicated over this event bus. Listing 4.12 shows the input stream definition for the *Ratings* event type for the topic *Event::Ratings*. Additionally, the maximum time span for which an event sent via this input stream remains valid is defined.

```
<?xml version="1.0" encoding="UTF-8" standalone=‘‘yes’’?>
<InStream>
  <Name>hubSchedule</Name>
  <EventType>Ratings</EventType>
  <Binding>
    <Broker>EventBus</Broker>
    <Destination>Event::Ratings</Destination>
    <DestinationType>TOPIC</DestinationType>
  </Binding>
  <MaxEventValidity unit=‘‘ms’’>1</MaxEventValidity>
</InStream>
```

Listing 4.12: The *Ratings* input stream definition.

Once the event type and the input stream are defined, and events are delivered via the *JMS* adapter, a *CEP* query is able to analyze events. Listing 4.13 displays the *EQL* query that calculates the item deviation. This is very similar to that of the *MySQL* implementation shown in Listing 4.8 on page 63. The only differences are the lack of conditions to limit the number of calculations, and the window range, which is only available in *EQL*. The windows implemented in this instance are tumbling windows. This means that on a daily basis, the events from the type *Ratings* are analyzed using this query. The two *Ratings* event types are then combined using the join on *UserID*.

```
SELECT
  r1.MovieID AS Item1,
  r2.MovieID AS Item2,
  AVG(r1.Rating - r2.Rating) AS ItemDeviation
FROM
  Ratings WINDOW( RANGE 1 DAY SLIDE 1 DAY ) AS r1,
  Ratings WINDOW( RANGE 1 DAY SLIDE 1 DAY ) AS r2
WHERE
  r1.UserID = r2.UserID AND
  r1.MovieID != r2.MovieID
GROUP BY
  r1.MovieID,
  r2.MovieID;
```

Listing 4.13: *CEP* query to produce a general preference deviation between items.

CEP queries are defined in *XML* files which include more information about the query and its environment. Listing 4.14 on page 68 shows the full *XML* definition for the query in Listing 4.8 on page 63. The name attribute of the query is self-explanatory and must

be used for the query's file name also, in this case *ItemDeviation.xml*. The event type defines the structure of the output and must be defined prior to usage. Unless directly stored into a table, a *SQL* query does not have to follow a predefined structure for its output. A *ConsoleOutput* is only used for debugging purposes in the IDE. The *JMSOutput* defines whether or not the query is active and then publishes complex events on the event bus. The *Binding* block defines the event bus instance on which the events are published. Any existing dependencies, in terms of which event types are required, are defined in the *Dependencies* block. The minimum interval between two events of the same type is defined in *MinimumHeartbeatInterval*.

```
1  <?xml version="1.0" encoding="UTF-8" standalone=‘‘yes’’?>
2  <Query>
3    <Name>ItemDeviation</Name>
4    <EventType>ItemDeviation</EventType>
5    <QueryString>SELECT &#xD;
6    r1.MovieID AS Item1, &#xD;
7    r2.MovieID AS Item2, &#xD;
8    AVG(r1.Rating - r2.Rating) AS ItemDeviation&#xD;
9  FROM &#xD;
10   Ratings WINDOW(RANGE 1 HOUR SLIDE 1 HOUR) AS r1&#xD;
11   JOIN &#xD;
12   Ratings WINDOW(RANGE 1 HOUR SLIDE 1 HOUR) AS r2&#xD;
13   ON (r1.UserID = r2.UserId)&#xD;
14 WHERE &#xD;
15   r1.MovieID != r2.MovieID&#xD;
16 GROUP BY &#xD;
17   r1.MovieID, r2.MovieID;</QueryString>
18   <ConsoleOutput>true</ConsoleOutput>
19   <JMSOutput>true</JMSOutput>
20   <Binding>
21     <Broker>EventBus</Broker>
22     <Destination>Event::ItemDeviation</Destination>
23     <DestinationType>TOPIC</DestinationType>
24   </Binding>
25   <Dependencies>
26   <Dependency type=‘‘SRC’’ name=‘‘Ratings’’ project=‘‘
       GroupLensDemoCEPProject’’/>
27   </Dependencies>
28   <MinimumHeartbeatInterval unit=‘‘ms’’>1</
       MinimumHeartbeatInterval>
```

```
29 </Query>
```

Listing 4.14: Complete *XML* version of the ItemDeviation query.

The query produces events of the type *ItemDeviation*, which also needs to be declared prior to usage. The definition is shown in Listing 4.15.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:eda="
       http://namespaces.softwareag.com/EDA/Event" xmlns:p="http://
       namespaces.softwareag.com/EDA" targetNamespace="http://
       namespaces.softwareag.com/EDA" elementFormDefault=‘‘qualified’’
       >
3    <xs:import namespace="http://namespaces.softwareag.com/EDA/Event
         " schemaLocation="Event/Envelope.xsd"/>
4    <xs:element name=‘‘ItemDeviation’’ type="p:ItemDeviationType"
         substitutionGroup="eda:Payload"/>
5    <xs:complexType name=‘‘ItemDeviationType’’>
6      <xs:sequence>
7        <xs:element name="Item1" type="xs:long"/>
8        <xs:element name="Item2" type="xs:long"/>
9        <xs:element name=‘‘Deviation’’ minOccurs="0" type="xs:double"
             />
10     </xs:sequence>
11   </xs:complexType>
12 </xs:schema>
```

Listing 4.15: The ItemDeviation event type.

The prediction query used for *MySQL*, as shown in Listing 4.10 on page 65, contains a *[NOT] IN* function. This function is currently not available in *Software AG's CEP*, although it is defined in *EQL*'s *BNF* grammar. The grammar suggests the use of *NOT IN* with a predefined set of elements, but that approach is not appropriate for this example. Therefore, it needs to be replaced with another construct. A *NOT IN* can be replaced with three different constructs: *NOT EXISTS*, *LEFT OUTER JOIN*, and *MINUS* or *EXCEPT*. Listing 4.16 shows an example query using *NOT IN* to demonstrate its possible replacements.

```
1 SELECT
2   *
3 FROM
4   Movies
```

```
5  WHERE
6    MovieID
7    NOT IN
8    (SELECT DISTINCT MovieID
9    FROM Ratings
10   WHERE UserID = 1);
```

Listing 4.16: Example of a use for *NOT IN* in *MySQL*.

1. *NOT IN* can be replaced with *NOT EXISTS* – see Listing 4.17. Here the sub-select needs to return something, in case the line does not appear in the result set. *EQL* does not support *NOT EXISTS*.

```
1  SELECT
2    *
3  FROM
4    Movies AS m
5  WHERE
6    NOT EXISTS
7    (SELECT MovieID
8    FROM Ratings AS r
9    WHERE
10     r.UserID = 1 AND
11     r.MovieID = m.MovieID);
```

Listing 4.17: Replacement of *NOT IN* with *NOT EXISTS* in *MySQL*.

2. *NOT IN* can be replaced with *LEFT OUTER JOIN* – see Listing 4.18. Here, the lines are only printed when a match is not found on the right side. This is not limited to the left side, but it is more intuitive in western culture to look from left to right. *LEFT OUTER JOIN* or any outer join is not supported by *EQL*.

```
1  SELECT
2    m.MovieID,
3    m.Title,
4    m.Genres
5  FROM
6    Movies AS m
7    LEFT OUTER JOIN
8    (SELECT * FROM Ratings WHERE UserID = 1) AS r
9    ON (m.MovieID = r.MovieID)
10 WHERE
```

```
11      r.MovieID is NULL;
```

Listing 4.18: Replacement of *NOT IN* with *LEFT OUTER JOIN* in *MySQL*.

3. *NOT IN* can be replaced with *EXCEPT* or *MINUS*, depending on the implementation – see Listing 4.19. There are two ways to realize this. Here, the movies' table is joined with a list of movies which the user with *UserId* 1 has not yet rated. *EXCEPT* is the keyword used by *PostgreSQL*; *Oracle* and *EQL* use *MINUS* as their keyword for the same functionality.

```
1  SELECT
2    ``m''.``MovieID'',
3    ``m''.``Title'',
4    ``m''.``Genres''
5  FROM
6    ``Movies'' AS ``m''
7    JOIN
8    (SELECT DISTINCT ``MovieID'' FROM ``Ratings''
9    EXCEPT
10   SELECT DISTINCT ``MovieID'' FROM ``Ratings'' WHERE ``UserID'' =
       1) AS ``newmovies''
11   ON (``m''.``MovieID'' = ``newmovies''.``MovieID'');
```

Listing 4.19: Replacement of *NOT IN* with *EXCEPT* in *PostgreSQL*.

To create the prediction for a certain user, the *UserID* was hard coded in the *MySQL* (Listing 4.10 on page 65) versions of this query. Here, the incoming ratings can be used to determine users for which predictions will be made. A sub-select is used to determine the users' average movie rating in the *MySQL* query. Sub-selects, although defined in the grammar, are not yet available in *EQL*, but can be replaced with other queries. For this reason, the users' average movie rating needs to have its own event type and query, as shown in Listing 4.20, and Listing 4.21 on page 72. The average ratings are calculated every day for the ratings a user has given during that time span, as is the item deviation. Therefore, the results are much more dynamic than the original *Apache Mahout* implementation and the database version. However, they also encounter the problem that users can only receive results if they have rated a movie on the previous day.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:eda
     ="http://namespaces.softwareag.com/EDA/Event" xmlns:p="http://
     namespaces.softwareag.com/EDA" targetNamespace="http://
     namespaces.softwareag.com/EDA" elementFormDefault=``qualified''
     >
```

```
3   <xsd:import namespace="http://namespaces.softwareag.com/EDA/
        Event" schemaLocation="Event/Envelope.xsd"/>
4   <xsd:element name=‘‘AverageRating’’ type="p:AverageRatingType"
        substitutionGroup="eda:Payload"/>
5   <xsd:complexType name=‘‘AverageRatingType’’>
6     <xsd:sequence>
7       <xsd:element name=‘‘UserID’’ type="xsd:long"/>
8       <xsd:element name=‘‘AverageRating’’ minOccurs="0" type="
            xsd:double"/>
9     </xsd:sequence>
10  </xsd:complexType>
11 </xsd:schema>
```

Listing 4.20: The AveragePrediction event type.

```
1  SELECT
2    r.UserID AS UserID,
3    AVG(rwin.Rating) AS AverageRating
4  FROM
5    Ratings WINDOW(RANGE 1 DAY) AS r
6  GROUP BY
7    r.UserID;
```

Listing 4.21: *CEP* query to calculate the average rating of users.

Based on the item deviation and average ratings, and using the *MINUS* function as shown in Listing 4.19 on page 71, the *Slope One* algorithm can be written as seen in Listing 4.22. It only receives events from the previous queries and therefore, inherits their window size and tumbling nature. The *MINUS* portion of the query is maximized to ensure the creation of exact copies of the events that are to be removed.

```
1  SELECT
2    ar.UserID AS UserID,
3    id.Item1 AS Item,
4    ar.AverageRating + AVG(id.ItemDeviation) AS Prediction,
5  FROM
6    ItemDeviation AS id
7    JOIN
8    AverageRating AS ar
9  GROUP BY
10   ar.UserID,
```

```
  id.item1,
  ar.AverageRating
MINUS
SELECT
  ar.UserID AS UserID,
  id.Item1 AS Item,
  ar.AverageRating + AVG(id.ItemDeviation) AS Prediction,
FROM
  ItemDeviation AS id
  JOIN
  AverageRating AS ar
  JOIN
  Ratings WINDOW(RANGE 1 DAY) AS rwin
  ON (ar.UserID = rwin.UserID AND id.Item1 = rwin.MovieID)
GROUP BY
  ar.UserID,
  id.item1,
  ar.AverageRating;
```

Listing 4.22: *CEP* query to produce a preference prediction for user 1.

Also, the *Slope One* prediction query needs an event type which is shown in Listing 4.23.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:eda="
   http://namespaces.softwareag.com/EDA/Event" xmlns:p="http://
   namespaces.softwareag.com/EDA" targetNamespace="http://
   namespaces.softwareag.com/EDA" elementFormDefault=‘‘qualified’’
   >
  <xs:import namespace="http://namespaces.softwareag.com/EDA/Event
     " schemaLocation="Event/Envelope.xsd"/>
  <xs:element name=‘‘Prediction’’ type="p:PredictionType"
     substitutionGroup="eda:Payload"/>
  <xs:complexType name=‘‘PredictionType’’>
    <xs:sequence>
      <xs:element name=‘‘UserID’’ type="xs:long"/>
      <xs:element name=‘‘Item’’ type="xs:long"/>
      <xs:element name=‘‘Prediction’’ minOccurs="0" type="xs:double
         "/>
    </xs:sequence>
```

```
11    </xs:complexType>
12  </xs:schema>
```

Listing 4.23: The Prediction event type.

The query lacks the sorting mechanism as described in the conceptual description of the *CEP* queries. A decision making subscriber of the prediction events would therefore also be required to sort the events by the predicted rating. This could be fixed by using the *UDO* function *topkanalyzer*. Additionally, reducing the number of events to those sent during one day would render the results more dynamic (but less precise).

4.2.5 CEP Queries Using Database Connectivity

CEP queries can be configured to access a database table directly, as if they were streams of events. *Software AG's CEP* allows connections to *DB2*, *Oracle*, *Apache Derby* and *Microsoft SQL* servers. To gain access to a database, a source file needs to be set up for every database table that is being used inside the *CEP*. Listing 4.24 shows this configuration file for a table containing the movie information of the Grouplens Project data set. Here, the table is stored in an *Apache Derby* database. The name of the database configuration must be used in an element in the *XML* file, as well as for the configuration file's name, here *Movies.dbsource*. The regular database structure is stored in the self-explanatory elements *Table*, *Schema*, *Catalog*, and *Username*. The element *DtpProfile* expects the name of a predefined database profile created in the Eclipse project Data Tools Platform. It acquires the information from the profile to fill in the complex element *ConnectionProperties*. The final element, *Caching*, defines in how the table is cached for future use. The possible values are *<None/>* for no caching; *<Infinite/>* for one time caching; and a complex block defining start time and interval size for periodic caching, as shown in the lines 24 through 26.

```
1  <?xml version="1.0" encoding="UTF-8" standalone=''yes''?>
2  <DatabaseSource>
3    <Name>Movies</Name>
4    <Table>MOVIES</Table>
5    <Schema>APP</Schema>
6    <Catalog>not_enabled</Catalog>
7    <Username>user</Username>
8    <DtpProfile>Derby</DtpProfile>
9    <ConnectionProperties>
10     <DriverClassPath>
11       C:/SoftwareAG/eclipse/v36/plugins/com.softwareag.wep.ext.
             derby.core_8.2.2.0011-0133/derbyclient.jar
12     </DriverClassPath>
```

```
13      <DriverClass>
14        org.apache.derby.jdbc.ClientDriver
15      </DriverClass>
16      <JdbcConnectionUrl>
17        jdbc:derby://localhost:1527/sample
18      </JdbcConnectionUrl>
19      <ConnectionParameters></ConnectionParameters>
20      <Vendor>Derby</Vendor>
21      <Version>10.2</Version>
22    </ConnectionProperties>
23    <Caching>
24      <Start>2012-08-05T00:00:00.000+02:00</Start>
25      <TimeZoneId>Europe/Berlin</TimeZoneId>
26      <Interval unit=''d''>1</Interval>
27    </Caching>
28  </DatabaseSource>
```

Listing 4.24: Movies.dbsource file.

Once the database table's access is correctly defined, the table can be used in a query – just like an event type – by using its name. Therefore, it is obvious that a database table's configuration name cannot already be used as an input stream or query name, since this would lead to a non-resolvable name ambiguity. Listing 4.25 now shows how the *Slope One* query from Listing 4.22 on page 72 can be extended to access master data from a database. The database table is added in the *FROM* clause in line 11, and its content is used in line 5. In this case, the name of the movie is added to the outgoing complex event so it becomes more understandable for humans. Since the structure of the resulting events is different from the previous *Slope One* query, the event type needs to be rewritten or altered to suit this query.

```
1  SELECT
2    ar.UserID AS UserID,
3    id.Item1 AS Item,
4    ar.AverageRating + AVG(id.ItemDeviation) AS Prediction,
5    mv.TITLE AS Movie
6  FROM
7    ItemDeviation AS id
8    JOIN
9    AverageRating AS ar
10   JOIN
11   Movies AS mv
12   ON (id.Item1 = mv.movieid)
```

```
13 GROUP BY
14   ar.UserID,
15   id.item1,
16   ar.AverageRating
17 MINUS
18 SELECT
19   ar.UserID AS UserID,
20   id.Item1 AS Item,
21   ar.AverageRating + AVG(id.ItemDeviation) AS Prediction,
22   mv.Title AS Title
23 FROM
24   ItemDeviation AS id
25   JOIN
26   AverageRating AS ar
27   JOIN
28   Movies AS mv
29   ON (id.Item1 = mv.MovieID)
30   JOIN
31   Ratings WINDOW(RANGE 1 DAY) AS rwin
32   ON (ar.UserID = rwin.UserID AND id.Item1 = rwin.MovieID)
33 GROUP BY
34   ar.UserID,
35   id.item1,
36   ar.AverageRating;
```

Listing 4.25: *Slope One* prediction query using a database table.

This example shows how master data can be integrated into an existing query. By using a custom component, the data in the database can also be directly altered, according to information on the event bus. This enables the use of weights and cluster means that change over time.

4.3 Fire Detection System Based on Intelligent Data Fusion Technology

This section demonstrates the usage of pattern matching and *User-Defined Functions* to realize cognition in *CEP*. First, the general structure with which the entire system is realized in *CEP* will be introduced. Next, two of the components are utilized to demonstrate pattern matching and *User-Defined Functions* using Bao et al.'s *Fire Detection System Based on Intelligent Data Fusion Technology*.

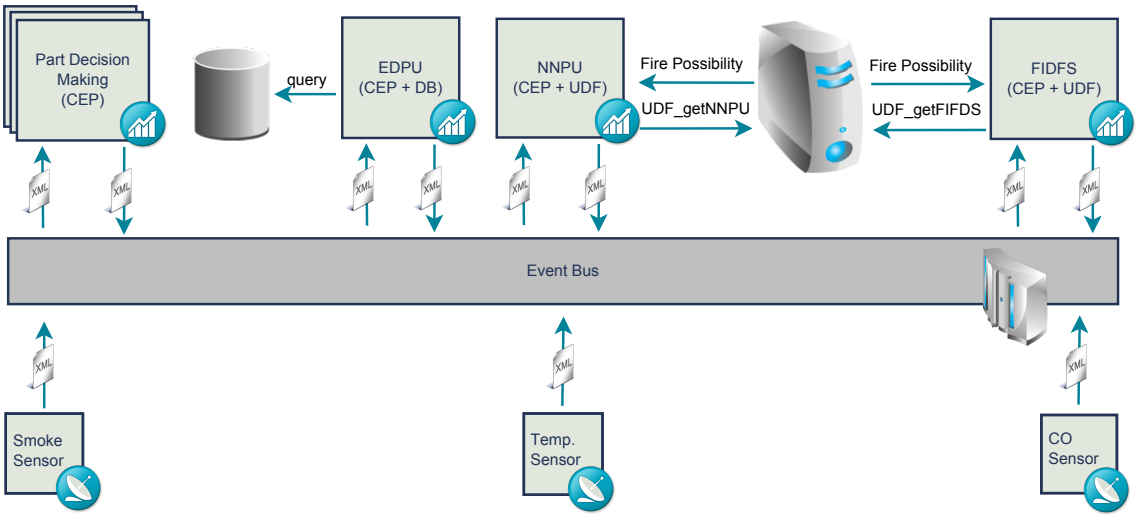

Figure 4.3: Realization of the fire detection of Bao et al. with *CEP*.

The fire detection algorithm, as described in section 2.3.2 on page 21, consists of three steps. Figure 2.7 on page 21 shows this division into three layers: the signal layer, the characteristic layer, and the decision layer. This structure can be adopted into the *CEP* realization. In the signal layer, three parallel components make partial decisions based on changes in the values in the sequential order of the input values. These changes are detected using the pattern matching. The *NNPU* has been realized using a *UDF*. Matrix multiplications can actually be realized in a *CEP* query. The realization of the learning neural network, on the other hand, could be realized in *UDF* or preferably in a custom component. In the custom component, the weights are stored as static variables and the end result can easily be utilized for back propagation. Once the learning is complete, the weights can be stored in the database and accessed by a *CEP* query. The characteristic layer's *EDPU* accesses an expert database and, therefore, could ideally be realized with access to a relational database. In this case, this has also been realized with a *UDF*. Last, the fuzzy inference system also accesses a *UDF*. Due to a lack of complete description of the specific implementation in Bao et al.'s paper, only the interface has been realized in this example. This structure used in the *CEP* realization is shown in Figure 4.3 on page 77.

The only event type required in this realization is *FireSensor.xsd* as shown in Listing 4.26 on page 78. Any sensor or query within the system only calculates the probability of a fire and it should always be trackable back to the sensors which sent the information. Since three sensors – for smoke particles, temperature, and carbon monoxide concentration – are combined into one set, they receive a group ID.

In this example, the practical capacity of the *IoT* is apparent A single fire sensor, e.g., a smoke sensor, is seen as a *VO* in the system. By aggregating the values from the remaining two sensors, a new and larger sensor – a *CVO* – is created.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
3    xmlns:eda="http://namespaces.softwareag.com/EDA/Event"
4    xmlns:p="http://namespaces.softwareag.com/EDA"
5    targetNamespace="http://namespaces.softwareag.com/EDA"
       elementFormDefault=‘‘qualified’’>
6    <xs:import namespace="http://namespaces.softwareag.com/EDA/Event
       " schemaLocation="Event/Envelope.xsd"/>
7    <xs:element name=‘‘FireSensor’’ type="p:FireSensorType"
       substitutionGroup="eda:Payload"/>
8    <xs:complexType name=‘‘FireSensorType’’>
9      <xs:sequence>
10       <xs:element name=‘‘SensorGroupID’’ minOccurs="0" type="xs:int
           "/>
11       <xs:element name=‘‘SensorValue’’ minOccurs="0" type="
           xs:double"/>
12     </xs:sequence>
13   </xs:complexType>
14 </xs:schema>
```

Listing 4.26: Event type definition for fire sensors.

To distinguish between all three sensor types, an input stream definition is needed for each type of sensor. Listing 4.27 shows the input stream definition for the carbon monoxide sensors, the *COSensor* input stream. The other input streams are similar, differing only in name, i.e. *SmokeSensor* and *TempSensor*.

```
1  <?xml version="1.0" encoding="UTF-8" standalone=‘‘yes’’?>
2  <InStream>
3    <Name>COSensor</Name>
4    <EventType>FireSensor</EventType>
5    <Binding>
6      <Broker>EventBus</Broker>
7      <Destination>Event::FireSensor</Destination>
8      <DestinationType>TOPIC</DestinationType>
9    </Binding>
10   <MaxEventValidity unit=‘‘ms’’>1</MaxEventValidity>
11 </InStream>
```

Listing 4.27: CO sensor input stream for the fire detection system.

4.3.1 Pattern Matching

The "part decision making" process in the signal layer is performed in two steps. First, two sequential events from the same sensor are analyzed for jumps. Listing 4.28 on page 79 shows the pattern used to receive the variation in the sensor value between two events, as used in equation 2.4 on page 22. One query exists per sensor type, and the *PARTITION* clause ensures that the values from only one sensor are analyzed. The difference in the sensor value between the two events is calculated in line 13.

```
1  SELECT
2    id AS SensorGroupID,
3    value AS SensorValue
4  FROM
5    COSensor
6  MATCHING
7  (
8    PARTITION BY SensorGroupID
9    MEASURES id INTEGER, value DOUBLE
10   PATTERN 'ab'
11   DEFINE
12     a DO id = SensorGroupID, value = SensorValue
13     b DO value = value - SensorValue
14 );
```

Listing 4.28: "Part decision" query for carbon monoxide sensors.

Equation 2.4 summed up the values of all three sensors in the same sensor group before subtracting it by a threshold value in equation 2.5. Both are realized in the second query as shown in Listing 4.29.

The first query returns complex events with a start time of the second incoming event. Given that a sensor sends a value every second, this query will return a value every second. The second query will, in turn, pick up the "part decision" from the three sensors with each passing second. If a sensor malfunctions due to fire damage, or vandalism or other problems, and discontinues sending events, further calculations become impossible. The queries will only run if an event of every input stream that is used arrives at the engine. Analysis for so-called non-events is necessary to manage this case.

```
1  SELECT
2    co.SensorGroupID AS SensorGroupID,
3    co.SensorValue + s.SensorValue + t.SensorValue - 0.5 AS
       SensorValue
4  FROM
5    COPartDecision AS co
```

```
6   JOIN
7   SmokePartDecision AS s
8   ON ( co.SensorGroupID = s.SensorGroupID )
9   JOIN
10  TempPartDecision AS t
11  ON ( co.SensorGroupID = t.SensorGroupID )
12 WHERE
13  co.SensorValue + s.SensorValue + t.SensorValue > 0.5;
```

Listing 4.29: "Part decision" query combining all three sensor types' "part decisions".

4.3.2 User-Defined Functions

The last method available in *Software AG's CEP* that is demonstrated in this thesis is the *User-Defined Function*. This method makes it possible to move complex calculations into a Java method that is also capable of storing information.

From the fire detection system, as described by Bao et al., the *NNPU* and the *FIDFS* must be realized in *UDF* since the methods described before are not sufficient. In this case, the *EDPU* is also realized in one *UDF*. The more practical method would be to use only a unit step function in *UDF*, and then, access the data from a relational database. In this case, only the *NNPU* is presented to illustrate the use of *UDF*s. The other two functions work similarly.

Listing 4.30 displays the class that contains the *User-Defined Function*s, but only the *NNPU* function's implementation is shown. The *getNNPUResult* method implements the *NNPU*, as described in Figure 2.8 on page 22, using the values from equations 2.10 and 2.11. It receives the values from the three sensors in a sensor group, performs two matrix multiplications with the given values, and returns a new probability for fire. The *UDF* is made available for use in a *CEP* query by the annotation *@UserDefinedFunction(name = "getNNPUResult")*. This is a static version using only the weights determined by Bao et al. A fully functional neural network that can be used to train the weights can also be implemented in a *UDF*.

```
1  package com.softwareag.DataFusionFireDetection;
2
3  import com.softwareag.wep.resource.udf.UserDefinedFunction;
4  import com.softwareag.wep.resource.udf.UserDefinedFunctions;
5
6  public class FireDetectionNNPU extends UserDefinedFunctions {
7
8    @UserDefinedFunction(name = ''unitStepFunction'')
```

```
9   public static Integer unitStepFunction(Double value) {
10    return (int) (Math.round(new Float(value) * 10) / 10.0);
11  }
12
13  /**
14   * Trained neural network to calculate the fire probability
15   * from values from smoke , temperature and CO sensors
16   * as user defined function.
17   * @param smoke The value of the smoke sensor [0.0 - 1.0].
18   * @param temperature The value of the temperature sensor [0.0
         - 1.0].
19   * @param coSignal The value of the CO sensor [0.0 - 1.0].
20   * @return The fire probability [0.0 - 1.0].
21   */
22  @UserDefinedFunction(name = ``getNNPUResult'')
23  public static Double getNNPUResult(Double smoke, Double
        temperature,
24      Double coSignal) {
25    Double w11 = 2.7998 * smoke - 0.1424 * temperature + 0.3836 *
          coSignal;
26    Double w12 = -.8670 * smoke + 2.0763 * temperature + 1.1122 *
          coSignal;
27    Double w13 = -.2689 * smoke + 2.6346 * temperature - 0.5426 *
          coSignal;
28
29    return 0.5442 * w11 - 2.1063 * w12 + 0.0204 * w13;
30  }
31
32  @UserDefinedFunction(name = ``getEDPUResult'')
33  public static Double getEDPUResult(Double smoke, Double
        temperature,
34      Double coSignal) {
35    [...]
36  }
37
38    @UserDefinedFunction(name = ``getFIDFSResult'')
39  public static Double getFIDFSResult(Double o1, Double o2) {
40    [...]
41  }
```

```
42 }
```

Listing 4.30: Excerpt from the *UDF* class for the fire detection system showing the *NNPU* implementation.

Since all the computation is outsourced to the *UDF*, the query used only needs to collect the values required for the function to work, and form the result according to the output event type. Listing 4.31 on page 83 illustrates how the individual sensor values are collected and forwarded to the *UDF* by calling *UDF_getNNPUResult*. The *EDPU* and *FIDFS* work accordingly and call their respective functions, i.e., *UDF_getEDPUResult* and *UDF_getFIDFSResult*.

```
1  SELECT
2    cos.SensorGroupID AS SensorGroupID,
3    UDF_getNNPUResult(ss.SensorValue , ts.SensorValue , cos.
       SensorValue) AS SensorValue
4  FROM
5    COSensor AS cos
6    JOIN
7    SmokeSensor AS ss
8    ON ( cos.SensorGroupID = ss.SensorGroupID )
9    JOIN
10   TempSensor AS ts
11   ON ( cos.SensorGroupID = ts.SensorGroupID );
```

Listing 4.31: Query to initialize neural network fire detection, NNPU.ceq.

In conclusion, every step in the fire detection system only realizes a single aspect of the entire system. None of the presented components actually trigger an action like a fire alarm signal or send information to emergency services. A separate decision program is still required to subscribe to the results of one or more of the steps in the system via the event bus.

4.4 Summary

Different methods to realize cognitive algorithms are demonstrated with two example algorithms. These algorithms were selected because of their real-life usage, and their relationship to the concept of the *IoT*. First, the connection to the event bus via a *JMS* adapter is illustrated. Then, the *Apache Mahout* implementation of the *Slope One* algorithm is compared to a database implementation and two variants of the *CEP* implementation of the same algorithm. Last, the *Fire Detection Based on Intelligent Data Fusion Technology* algorithm is utilized to demonstrate the realization using pattern matching and *User-Defined Functions*.

5 SUMMARY, CONCLUSIONS AND RECOMMENDATIONS

5.1 Summary

The background chapter introduces the concepts required for this thesis, which are *Complex Event Processing*, *Internet of Things*, and cognitive technology. Each is briefly discussed. A broad overview of *CEP* is given, as well as its specific application in *Software AG*'s implementation. The *Internet of Things* is also presented in two ways; first to provide a basic foundational understanding, and then as it is conceptualized by the *iCore* project. Initially, some basic cognitive algorithms are presented to illustrate their functions. Then, two specific algorithms, which are used in real life, and are related to the concept of *IoT*, are described in full. These are the *Slope One* algorithm and the *Fire Detection Based on Intelligent Data Fusion Technology*.

The next chapter demonstrates ways to realize cognition using *CEP*. It begins with a presentation on how cognition is realized in general. Within the *CEP*, various functions are available to realize the cognitive algorithms, with advantages and disadvantages for various realization issues. Finally, a two-step method that assists in selecting the best functions for a given scenario is presented.

The realization chapter demonstrates a full realization of cognitive technology in *CEP*. The basic implementation of the *JMS* is shown, demonstrating how to serve the event bus with events. The two specific algorithms from the background section are then used to demonstrate the different functions in *CEP*. The *Slope One* algorithm is used to compare the *CEP* realization with its real-life implementation, both in *Apache Mahout* and with a database implementation. This algorithm is used to demonstrate a pure *CEP* query solution, as well as a solution using a database for master data. At the end of the chapter, the *Fire Detection Based on Intelligent Data Fusion Technology* algorithm is utilized to demonstrate its realization using pattern matching and *User-Defined Function*s.

5.2 Conclusions

Complex Event Processing provides many means that can be used to realize cognitive algorithms. By combining a variety of different features, all algorithms are able to be

implemented. After breaking an algorithm down to its smallest components, each component can be realized with the best-aligned *CEP* feature, as demonstrated on a set of basic machine learning algorithms.

The different standard features possess different advantages and disadvantages. The *EQL* queries are not capable of working with complex data structures, but are perfect for monitoring and aggregating events in order to reduce the amount of data for subsequent steps. Furthermore, an *EQL* query is the base for other standard features. For special sorts of aggregations, the *UDA*s can be implemented and accessed in an *EQL* query. Mathematical functions that exceed the capacity of the standard functions in *EQL*, or are not implemented in the standard functions, can be performed in custom *UDF*s. *UDO*s and totally custom-implemented event processing components complete the array of features. They both allow the definition of highly complex custom event processing mechanisms.

The two real-life example algorithms have proven to be implementable using the standard *CEP* functionality, following the suggested method. The *Slope One* classification algorithm fits a possible use case in the "Smart City" domain of *IoT*. Bao et al.'s *Fire Detection System Based on Intelligent Data Fusion Technology* can be used in the "Smart Home" and "Smart Business" domains. Combined, these examples are applicable to most *IoT* domains as defined in the *iCore* project.

Use cases and scenarios in *IoT* are diverse. This integration of cognitive technology into the *CEP* has shown that this combination can be used to realize *IoT* use cases. The scalable analysis of data inside *CEP* paired with the dynamic analysis by cognitive algorithms is, in fact, an ideal combination.

5.3 Outlook

In this thesis, the general possibilities for realizing *IoT* projects using cognition in *CEP* were analyzed. The specific use cases where this combination of technology could be applied are numerous. A thorough realization and analysis of the use cases that are presented in the *iCore* project should be done next.

Generally, the combination of the presented technology, along with a framework of cognitive algorithms, should be analyzed. Through the use of custom event processing components, a connection to the *Apache Mahout* project could be realized, enabling the utilization of the algorithms that use *Apache Hadoop*. The feasibility and usability of such a combination has yet to be explored.

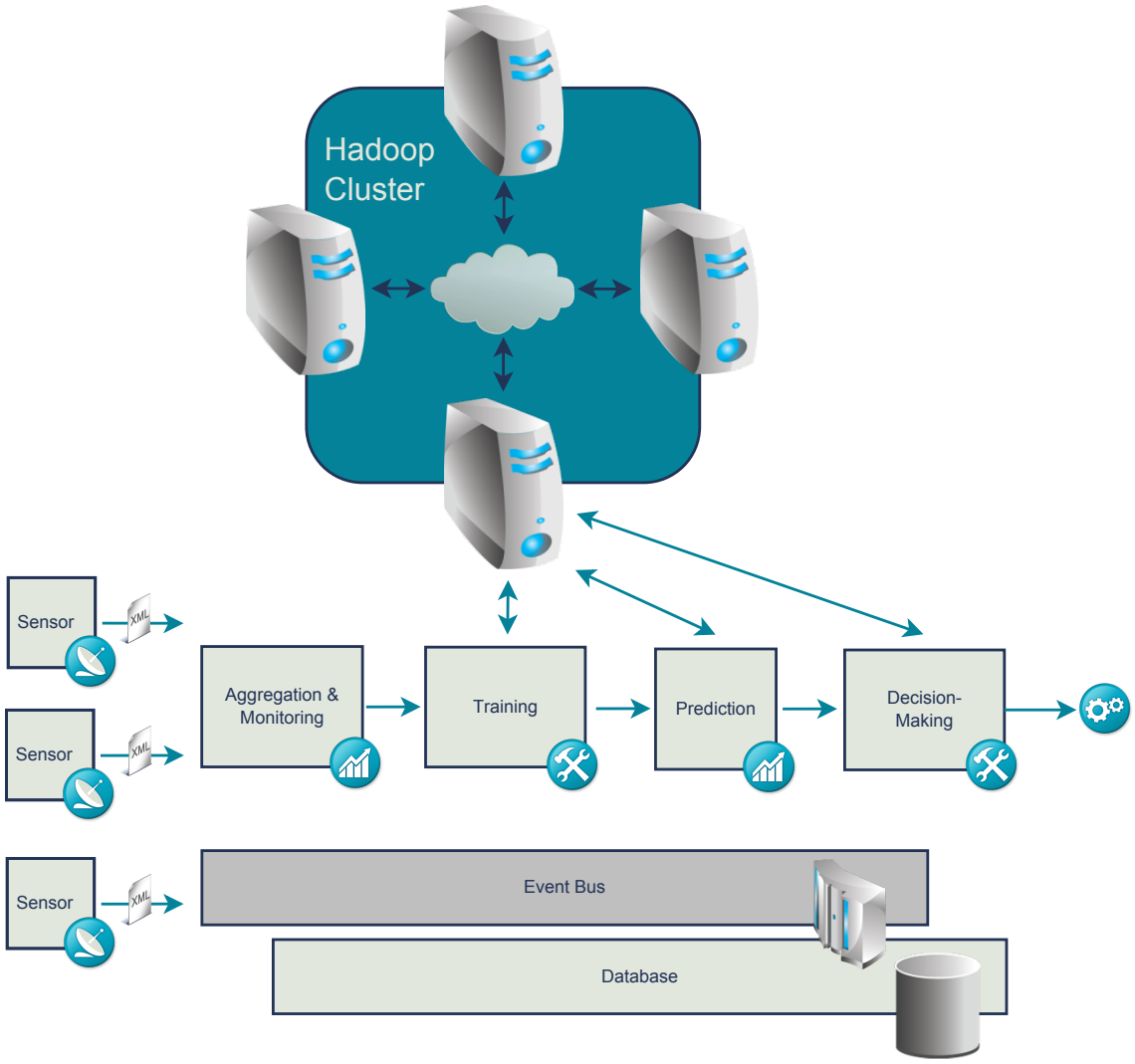

Figure 5.1: Possible combination of a distributed computation into *CEP*.

REFERENCES

[Ashton, 2009] Ashton, K. (2009). That 'internet of things' thing. Online, RFID Journal, http://www.rfidjournal.com/article/view/4986. Last access 13 Aug 2012.

[Atkinson et al., 1995] Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., and Zdonik, S. (1995). The object-oriented database system manifesto. Online, Computer Science Department of the Carnegie Mellon University, http://www.cs.cmu.edu/∼clamen/OODBMS/Manifesto/htManifesto/Manifesto.html. Last access 13 Aug 2012.

[Bao et al., 2003] Bao, H., Li, J., Zeng, X.-Y., and Zhang, J. (2003). A fire detection system based on intelligent data fusion technology. In *Proceedings of the Second International Conference on Machine Learning and Cybernetics*.

[Barth, 2009] Barth, D. (2009). Official google blog: The bright side of sitting in traffic: Crowdsourcing road congestion data. Online, Google Blog, http://googleblog.blogspot.com/2009/08/bright-side-of-sitting-in-traffic.html. Last access 13 Aug 2012.

[Curry, 2005] Curry, E. (2005). Message-oriented middleware. In Mahmoud, Q. H., editor, *Middleware for Communications*, chapter 1. John Wiley & Sons, Ltd.

[Etzion and Niblett, 2010] Etzion, O. and Niblett, P. (2010). *Event Processing in Action*. Manning Publications Company.

[Gamma et al., 1993] Gamma, E., Helm, R., Johnson, R. E., and Vlissides, J. M. (1993). Design patterns: Abstraction and reuse of object-oriented design. In *ECOOP*, pages 406–431.

[Honan, 2007] Honan, M. (2007). Apple unveils iphone — macworld. Online, MacWorld, http://www.macworld.com/article/54769/2007/01/iphone.html. Last access 13 Aug 2012.

[iCore Project, 2011a] iCore Project (2011a). icore project. Online, http://www.iot-icore.eu/. Last access 13 Aug 2012.

[iCore Project, 2011b] iCore Project (2011b). icore project consortium. Online, http://www.iot-icore.eu/consortium. Last access 01 Jun 2013.

[Lemire and MacLachlan, 2005] Lemire, D. and MacLachlan, A. (2005). Slope one predictors for online rating-based collaborative filtering. Technical report.

[Lepekhin, 2004] Lepekhin, E. (2004). Trees in sql databases. Online, The Code Project, http://www.codeproject.com/Articles/8355/Trees-in-SQL-databases. Las taccess 13 Aug 2012.

[Luckham, 2002] Luckham, D. C. (2002). *The power of events: an introduction to complex event processing in distributed enterprise systems*. Addison-Wesley, Boston.

[MacManus, 2011] MacManus, R. (2011). Cisco: 50 billion things on the internet by 2020 [infographic]. Online, http://www.readwriteweb.com/archives/cisco_50_billion_things_on_the_internet_by_2020.php. Last access 13 Aug 2012.

[Mattern and Flörkemeier, 2010] Mattern, F. and Flörkemeier, C. (2010). Vom Internet der Computer zum Internet der Dinge. *Informatik Spektrum*, 33(2):107–121.

[MIT AUTO-ID labs, 2008] MIT AUTO-ID labs (2008). AUTO-ID Labs at MIT. Online, MIT, http://autoid.mit.edu/. Last access 13 Aug 2012.

[OECD, 2012] OECD (2012). Machine-to-machine communications: Connecting billions of devices. Technical Report OECD Digital Economy Papers, No. 192, OECD Publishing, http://dx.doi.org/10.1787/5k9gsh2gp043-en.

[Prodanov, 2011] Prodanov, M. (2011). Xml integration into an sql-based cep engine. Master's thesis, Technische Universität Darmstadt.

[RTM Realtime Monitoring GmbH, 2010] RTM Realtime Monitoring GmbH (2010). *RTM Analyzer Tutorial*. RTM Realtime Monitoring GmbH, version 1.4 edition.

[Software AG, 2011] Software AG (2011). *Optimizing BPM and System Resources with BAM webMethods Optimize User's Guide Version 8.2*. Software AG.

[ten Hompel, 2011] ten Hompel, M. (2011). Das Internet der Dinge. Online, Fraunhofer IML, http://www.internet-der-dinge.de/. Last access 13 Aug 2012.

[van Kranenburg, 2008] van Kranenburg, R. (2008). The internet of things, a critique of ambient technology and the all-seeing network of rfid. Technical report, Institute of Network Cultures.

[Volk, 2002] Volk, R. (2002). More trees & hierarchies in sql. Online, SQLTeam.com, http://www.sqlteam.com/article/more-trees-hierarchies-in-sql. Last access 13 Aug 2012.

[Williams, 2011] Williams, C. (2011). Apple iphone tracks users' location in hidden file - telegraph. Online, Telegraph, http://www.telegraph.co.uk/technology/apple/8464122/Apple-iPhone-tracks-users-location-in-hidden-file.html. Last access 13 Aug 2012.

APPENDIX

5.4 BNF for EQL Grammar

This is the BNF form of the EQL as shown in the RTM Analyzer Tutorial [RTM Realtime Monitoring GmbH, 2010].

TOKENS
<DEFAULT> SKIP : {
“”
| “\t”
| “\r”
| “\n”
}

<DEFAULT> TOKEN :{
<K_ABS: “ABS”>
| <K_ALL: “ALL”>
| <K_AND: “AND”>
| <K_ANY: “ANY”>
| <K_AS: “AS”>
| <K_AVG: “AVG”>
| <K_BETWEEN: “BETWEEN”>
| <K_BY: “BY”>
| <K_CASE: “CASE”>
| <K_COUNT: “COUNT”>
| <K_CROSS: “CROSS”>
| <K_DAY: “DAY”>
| <K_DAYS: “DAYS”>
| <K_DEFAULT: “DEFAULT”>
| <K_DEFINE: “DEFINE”>
| <K_DISTINCT: “DISTINCT”>
| <K_DO: “DO”>
| <K_DURATION: “DURATION”>
| <K_END: “END”>
| <K_ELSE: “ELSE”>

| <K_EXCEPT: “EXCEPT”>
| <K_EXISTS: “EXISTS”>
| <K_FROM: “FROM”>
| <K_FULL: “FULL”>
| <K_GROUP: “GROUP”>
| <K_HOUR: “HOUR”>
| <K_HOURS: “HOURS”>
| <K_HAVING: “HAVING”>
| <K_IS: “IS”>
| <K_IN: “IN”>
| <K_INNER: “INNER”>
| <K_INTERSECT: “INTERSECT”>
| <K_JOIN: “JOIN”>
| <K_LEFT: “LEFT”>
| <K_LIKE: “LIKE”>
| <K_MAX: “MAX”>
| <K_MAX: “MAX”>
| <K_MATCHING: “MATCHING”>
| <K_MEASURES: “MEASURES”>
| <K_MIN: “MIN”>
| <K_MINUTE: “MINUTE”>
| <K_MINUTES: “MINUTES”>
| <K_MINUS: “MINUS”>
| <K_NATURAL: “NATURAL”>
| <K_NOT: “NOT”>
| <K_NOW: “NOW”>
| <K_NULL: “NULL”>
| <K_ON: “ON”>
| <K_OR: “OR”>
| <K_OUTER: “OUTER”>
| <K_PARTITION: “PARTITION”>
| <K_PATTERN: “PATTERN”>
| <K_PRIOR: “PRIOR”>
| <K_RANGE: “RANGE”>
| <K_RIGHT: “RIGHT”>
| <K_ROWS: “ROWS”>
| <K_SECOND: “SECOND”>
| <K_SECONDS: “SECONDS”>
| <K_SELECT: “SELECT”>
| <K_SHIFT: “SHIFT”>
| <K_SLIDE: “SLIDE”>
| <K_STDDEV: “STDDEV”>

```
| <K_STDDEV_POP: “STDDEV_POP”>
| <K_STDDEV_SAMP: “STDDEV_SAMP”>
| <K_SUM: “SUM”>
| <K_THEN: “THEN”>
| <K_UNBOUNDED: “UNBOUNDED”>
| <K_UNION: “UNION”>
| <K_USING: “USING”>
| <K_VARIANCE: “VARIANCE”>
| <K_VAR_POP: “VAR_PO”>
| <K_VAR_SAMP: “VAR_SAMP”>
| <K_WHEN: “WHEN”>
| <K_WHERE: “WHERE”>
| <K_WINDOW: “WINDOW”>
| <K_WITHIN: “WITHIN”>
}

<DEFAULT> TOKEN : {
<S_NUMBER: <FLOAT> | <FLOAT> ([“e”,“E”] ([“-”,“+”])? <FLOAT>)?>
| <#FLOAT: <INTEGER> | <INTEGER> (“.” <INTEGER>)? | “.” <INTEGER>>
| <#INTEGER: (<DIGIT>)+>
| <#DIGIT: [“0”-“9”]>
}

<DEFAULT> SPECIAL : {
<LINE_COMMENT: “–” ( [“\r”,“\n”])*>
| <MULTI_LINE_COMMENT: “/*” ( [“*”])* “*” (“*” | [“*”,“/”] ( [“*”])* “*”)* “/”>
}

<DEFAULT> TOKEN : {
<S_IDENTIFIER: (<LETTER>)+ (<DIGIT> | <LETTER> | <SPECIAL_CHARS>)*>
| <#LETTER: [“a”-“z”,“A”-“Z”]>
| <#SPECIAL_CHARS: “$” | “_”>
| <S_BIND: “:” <S_IDENTIFIER> (“.” <S_IDENTIFIER>)?>
| <S_CHAR_LITERAL: “\“ ( [“\”])* “\” (“\” ( [“\”])* “\”)*>
| <S_QUOTED_IDENTIFIER: “\”’ ( [“\n”,“\r”,“\”])* “\”>
}

NON-TERMINALS
SqlStatements := ( SqlStatement )+
SqlStatement := ( SelectStatement )
```

TableColumn := ObjectName (“.” ObjectName (“.” ObjectName)?)?
TableColumnList := TableColumn (“,” TableColumn)*
ObjectName := <S_IDENTIFIER>
| <S_QUOTED_IDENTIFIER>
Relop := “=”
| “!=”
| “#”
| “<>”
| “>”
| “>=”
| “<”
| “<=”
TableReference := ObjectName (“.” ObjectName)?
NumOrID := (<S_IDENTIFIER> | (“+” | “-”)? <S_NUMBER>)
SelectStatement := SelectWithoutOrder “;”
SelectWithoutOrder := “SELECT” (“ALL” | “DISTINCT”)? SelectList FromClause (SetCondition |
WhereClause)? (GroupByClause)? (SetClause)?
SelectList := “*”
| SelectItem (“,” SelectItem)*
SelectItem := (SelectStar | SqlSimpleExpression (Alias)?)
Alias := (“AS”)? <S_IDENTIFIER>
SelectStar := ObjectName “.” (ObjectName “.”)? “*”
FromClause := “FROM” FromItem (“,” FromItem)*
FromItem := ((TableReferenceOrSubquery (JoinedTable)*) | (“(” FromItem JoinedTable “)”))
TableReferenceOrSubquery := TableReference TemporalClause (Alias)? (MatchingClause)?
| “(” SubQuery “)” TemporalClause Alias (MatchingClause)?
WindowClause := “WINDOW” “(” (“PARTITION” “BY” TableColumnList)? (“ROWS” (<S_NUMBER> |
“UNBOUNDED”) (“SLIDE” <S_NUMBER>)? | “RANGE” (TimeExpression | “UNBOUNDED” | “NOW”)
(“SLIDE” TimeExpression)?) “)”
TimeExpression := <S_NUMBER> ((“SECOND” | “SECONDS”) | (“MINUTE” | “MINUTES”) | (
“HOUR” | “HOURS”) | (“DAY” | “DAYS”))?
JoinedTable := ((“CROSS” “JOIN” TableReferenceOrSubquery) | (“NATURAL” (“INNER” | (“LEFT” |
“RIGHT” | “FULL”) (“OUTER”)? | “UNION”)? “JOIN” TableReferenceOrSubquery) | ((“INNER” | (
“LEFT” | “RIGHT” | “FULL”) (“OUTER”)? | “UNION”)? “JOIN” TableReference-

OrSubquery (“ON”
SqlExpression | “USING” “(” TableColumnList “)”)?))
WhereClause := “WHERE” SqlExpression
GroupByClause := “GROUP” “BY” SqlExpressionList (“HAVING” SqlExpression)?
SetClause := (“UNION” (“ALL”)? | “INTERSECT” | (“MINUS” | “EXCEPT”)) ((
“(” SelectWithoutOrder
“)”) | SelectWithoutOrder)
SqlExpression := SqlAndExpression (“OR” SqlAndExpression)*
SqlAndExpression := SqlUnaryLogicalExpression (“AND” SqlUnaryLogicalExpression)*
SqlUnaryLogicalExpression := (ExistsClause | (“NOT”)? SqlRelationalExpression)
ExistsClause := (“NOT”)? “EXISTS” “(” SubQuery “)”
SqlRelationalExpression := (“(” SqlExpressionList “)” | ((“PRIOR”)? SqlSimpleEx-
pression)) (
SqlRelationalOperatorExpression | (SqlInClause) | (SqlBetweenClause) | (SqlLikeClause) |
IsNullClause)?
SqlExpressionList := SqlSimpleExpressionOrPreparedCol (“,” SqlSimpleExpressionOr-
PreparedCol)*
SqlRelationalOperatorExpression := Relop (((“ALL” | “ANY”)? “(” SubQuery “)”) |
(“PRIOR”)?
SqlSimpleExpressionOrPreparedCol)
SqlSimpleExpressionOrPreparedCol := (SqlSimpleExpression)
SqlInClause := (“NOT”)? “IN” “(” (SqlExpressionList | SubQuery) “)”
SqlBetweenClause := (“NOT”)? “BETWEEN” SqlSimpleExpressionOrPreparedCol “AND”
SqlSimpleExpressionOrPreparedCol
SqlLikeClause := (“NOT”)? “LIKE” SqlSimpleExpressionOrPreparedCol
IsNullClause := “IS” (“NOT”)? “NULL”
SqlSimpleExpression := SqlMultiplicativeExpression (“+” SqlMultiplicativeExpression |
“-”
SqlMultiplicativeExpression | “||” SqlMultiplicativeExpression)*
SqlMultiplicativeExpression := SqlExpotentExpression (“*” SqlExpotentExpression | “/”
SqlExpotentExpression)*
SqlExpotentExpression := SqlUnaryExpression (“**” SqlUnaryExpression)*
SqlUnaryExpression := (“+” | “-”)? SqlPrimaryExpression
SqlPrimaryExpression := OuterJoinExpression
| AggregateFunc “(” ((“ALL” | “DISTINCT”) TableColumn | SqlExpression) “)”
| FunctionCall
| TableColumn
| “COUNT(*)”
| Constant
| “(” SqlExpression “)”
Constant := “NULL”
| <S_NUMBER>

| <S_CHAR_LITERAL >
ConstantList := Constant (“,” Constant)*
AggregateFunc := “AVG”
| “COUNT”
| “MAX”
| “MIN”
| “STDDEV_POP”
| (“STDDEV” | “STDDEV_SAMP”)
| “SUM”
| “VAR_POP”
| (“VARIANCE” | “VAR_SAMP”)
FunctionCall := “ABS” “(” SqlExpression “)”
| “CASE” (SimpleCaseCall | BooleanCaseCall)
SimpleCaseCall := SqlSimpleExpression (“WHEN” SqlSimpleExpression “THEN” Sql-
SimpleExpression)+
(“ELSE” SqlSimpleExpression)? “END”
BooleanCaseCall := (“WHEN” SqlExpression “THEN” SqlSimpleExpression)+ (“ELSE”
SqlSimpleExpression)? “END”
OuterJoinExpression := ObjectName (“.” ObjectName (“.” ObjectName)?)? “(” “+”
“)”
SubQuery := SelectWithoutOrder
MatchingClause := “MATCHING” “(” (“PARTITION” “BY” TableColumnList)? (
MeasureClasue)?
“PATTERN” <S_CHAR_LITERAL> (“DURATION” TimeExpression)? (“WITHIN”
TimeExpression)? (
DefineClause)? “)” (WindowClause)? Alias
MeasureClause := “MEASURES” MeasureItem (“,” MeasureItem)*
MeasureItem := ObjectName BasicDatatypeDeclaration (“DEFAULT” (<S_NUMBER>
|
<S_CHAR_LITERAL>))?
DefineClause := “DEFINE” (DefineItem)+
DefineItem := <#LETTER> ((“AS” SqlExpression)? (“DO” Action (“,” Action)*)?
Action := ObjectName “=” SqlSimpleExpression
TemporalClause := (WindowClause)? (ShiftClause)?
ShiftClause := “SHIFT” “BY” TimeExpression
SetCondition := “WHERE” (TableColumn “IN” “(” ConstantList “)” | “(” TableColumn-
List “)” “IN” “(” “(”
ConstantList “)” (“,” “(” ConstantList “)”)* “)”)) (“ALL” | “COUNT” <S_NUMBER>
| “MIN”
<S_NUMBER> | “MAX” <S_NUMBER> | (“NOT”)? “BETWEEN” <S_NUMBER>
“AND” <S_NUMBER>)

5.5 CEP Envelope Schema Definition

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="
       http://namespaces.softwareag.com/EDA/Event"
3    targetNamespace="http://namespaces.softwareag.com/EDA/Event"
         elementFormDefault="qualified">
4
5    <xsd:element name="Event">
6      <xsd:complexType>
7        <xsd:sequence>
8          <xsd:element name="Header" type="HeaderType" />
9          <xsd:element name="Body" minOccurs="0">
10           <xsd:complexType>
11             <xsd:sequence>
12               <xsd:element ref="Payload" />
13             </xsd:sequence>
14           </xsd:complexType>
15         </xsd:element>
16       </xsd:sequence>
17     </xsd:complexType>
18   </xsd:element>
19
20   <xsd:element name="Payload" abstract="true" />
21
22   <xsd:complexType name="HeaderType">
23     <xsd:all>
24       <xsd:element name="Start" type="SAGDateTimeType">
25         <xsd:annotation>
26           <xsd:documentation xml:lang="en">
27             Start date and time of the event.
28           </xsd:documentation>
29         </xsd:annotation>
30       </xsd:element>
31       <xsd:element name="End" type="SAGDateTimeType" minOccurs="0">
32         <xsd:annotation>
33           <xsd:documentation xml:lang="en">
34             End date and time of the event. The use of this field
                   is dependent on how the event is being used. If the
                   value is absent, the event receiving application may
                    set a default one, e.g. start time plus one
```

```
                millisecond.
35          </xsd:documentation>
36        </xsd:annotation>
37      </xsd:element>
38      <xsd:element name="Kind" type="SAGKindType" minOccurs="0"
           default="Event">
39        <xsd:annotation>
40          <xsd:documentation xml:lang="en">
41            Indicates whether the event is a new event (Event) or a
                 so-called  heartbeat (Heartbeat), which indicates
                the temporal progress of the stream.
42          </xsd:documentation>
43        </xsd:annotation>
44      </xsd:element>
45      <xsd:element name="Type" type="xsd:string" minOccurs="0">
46        <xsd:annotation>
47          <xsd:documentation xml:lang="en">
48            The name of the event type.
49          </xsd:documentation>
50        </xsd:annotation>
51      </xsd:element>
52      <xsd:element name="Version" type="EventTypeVersionType"
           minOccurs="0">
53        <xsd:annotation>
54          <xsd:documentation xml:lang="en">
55            The version of the event type with which the event
                instance is compatible. Users specify this value if
                they have chosen to support event type versioning.
                If the event instance is not versioned, the event
                type should not be versioned.
56          </xsd:documentation>
57        </xsd:annotation>
58      </xsd:element>
59      <xsd:element name="CorrelationID" type="xsd:string" minOccurs
           ="0">
60        <xsd:annotation>
61          <xsd:documentation xml:lang="en">
62            Unique identifier used to associate the event instance
                with other  event instances.
63          </xsd:documentation>
64        </xsd:annotation>
65      </xsd:element>
```

```
66      </xsd:all>
67    </xsd:complexType>
68
69    <xsd:simpleType name="SAGDateTimeType">
70      <xsd:annotation>
71        <xsd:documentation xml:lang="en">Structure for values of date
              /time information. All DateTime element need to conform
              to ISO 8601.
72        </xsd:documentation>
73      </xsd:annotation>
74      <xsd:restriction base="xsd:dateTime">
75        <xsd:pattern value=".+T.+(Z|([+]|\\-).+)" />
76      </xsd:restriction>
77    </xsd:simpleType>
78
79    <xsd:simpleType name="SAGKindType">
80      <xsd:restriction base="xsd:token">
81        <xsd:enumeration value="Event" />
82        <xsd:enumeration value="Heartbeat" />
83      </xsd:restriction>
84    </xsd:simpleType>
85
86    <xsd:simpleType name="EventTypeVersionType">
87      <xsd:restriction base="xsd:string">
88        <xsd:pattern value="[0-9]+(.[0-9]+)*" />
89        <xsd:minLength value="1" />
90      </xsd:restriction>
91    </xsd:simpleType>
92
93  </xsd:schema>
```