Big continuous data: dealing with velocity by composing event streams

Genoveva Vargas-Solar, Javier Alfonso Espinosa Oviedo, José-Luis Zechinelli-Martini

Big continuous data: dealing with velocity by composing event streams

Genoveva Vargas-Solar[1], Javier A. Espinosa Oviedo[1], José Luis Zechinelli Martini[2]

[1]CNRS, LIG LAFMIA

681 rue de la Passerelle BP 72

38402 Saint Martin d'Hères, France

[2] Fundación Universidad de las Américas Puebla, LAFMIA

Exhacienda Sta. Catarina Mártir s/n

72810 San Andrés Cholula, México

genoveva.vargas@imag.fr, javier.espinosa@imag.fr, joseluis.zechinelli@udlap.mx

Abstract. The rate at which we produce data is growing steadily, thus creating even larger streams of continuously evolving data. Online news, micro-blogs, search queries are just a few examples of these continuous streams of user activities. The value of these streams relies in their freshness and relatedness to ongoing events. Modern applications consuming these streams need to extract behaviour patterns that can be obtained by aggregating and mining statically and dynamically huge event histories. An *event* is the notification that a happening of interest has occurred. Event streams must be combined or aggregated to produce more meaningful information. By combining and aggregating them either from multiple producers, or from a single one during a given period of time, a limited set of events describing meaningful situations may be notified to consumers. Event streams with their volume and continuous production cope mainly with two of the characteristics given to Big Data by the 5V's model: volume & velocity. Techniques such as complex pattern detection, event correlation, event aggregation, event mining and stream processing, have been used for composing events. Nevertheless, to the best of our knowledge, few approaches integrate different composition techniques (online and post-mortem) for dealing with Big Data velocity. This chapter gives an analytical overview of event stream processing and composition approaches: complex event languages, services and event querying systems on distributed logs. Our analysis underlines the challenges introduced by Big Data velocity and volume and use them as reference for identifying the scope and limitations of results stemming from different disciplines: networks, distributed systems, stream databases, event composition services, and data mining on traces.

1 Introduction

The rate at which we produce data is growing steadily, thus creating even larger streams of continuously evolving data. Online news, micro-blogs, search queries are just a few examples of these continuous streams of user activities. The value of these streams relies in their freshness and relatedness to on-going events. Massive data streams that were once obscure and distinct are being aggregated and made easily accessible.

Modern applications consuming these streams require to extract behaviour patterns that can be obtained by aggregating and mining statically and dynamically huge event histories. An *event* is the notification that a happening of interest has occurred. Event streams are continuous flows of events stemming from one or several producers. They must be combined or aggregated to produce more meaningful information. By combining and aggregating them either from multiple producers, or from a single one during a given period of time, a limited set of events describing meaningful situations may be notified to consumers.

Event streams with their volume and continuous production cope mainly with two of the characteristics given to Big Data by the 5V's model: volume & velocity. Event-based systems have gained importance in many application domains, such as management and control systems, large-scale data dissemination, monitoring applications, autonomic computing, etc. Event composition has been tackled by several academic research and industrial systems. Techniques such as complex pattern detection, event correlation, event aggregation, event mining and stream processing, have been used for composing events. In some cases event composition is done on event histories (e.g. event mining) and in other cases it is done online as events are produced (e.g. event aggregation and stream processing). Nevertheless, to the best of our knowledge, few approaches integrate different composition techniques (online and post-mortem) for dealing with Big Data velocity and volume.

This chapter gives an analytical overview of event stream processing and composition approaches that can respond to the challenges introduced by Big Data volume and velocity. Examples of these approaches are complex event languages, services and event querying systems on distributed logs. Our analysis underlines the challenges introduced by Big Data velocity and volume and use them as reference for identifying the scope and limitations of results stemming from different disciplines: networks, distributed systems, stream databases, event composition services, and data mining on traces.

Accordingly, this chapter is organized as follows. Section 2 introduces the problem related to Big Data velocity by studying two main techniques: event histories and online event processing. It also describes target applications where data velocity is a key element. Section 3 gives an overview of existing event stream

models. It discusses the main principles for modelling event streams. Section 4 gives an overview of event composition techniques. It compares existing approaches for exploiting streams either by composing them or by applying analytics techniques. Finally, section 5 concludes the chapter and discusses big data velocity outlook.

2 Big data velocity issues

This section introduces the challenges associated with Big Data velocity. In particular it describes stream processing challenges and results that are enabling ways of dealing with Big Data velocity. First the section gives the general lines of stream processing and existing prominent systems. Then it discusses event histories, which provide a complementary view for dealing with continuous data produced in a producers/consumers setting. The notion of event histories can be seen as Big Data produced at high rates and that must be analysed taking into consideration their temporal and spatial features. Finally, the section describes target applications families where Big Data velocity acquires particular importance.

2.1 Stream processing and velocity

Stream processing is a programming paradigm that processes continuous event (data) streams. They arise in telecommunications, health care, financial trading, and transportation, among other domains. Timely analysis of such streams can be profitable (in finance) and can even save lives (in health care). In the streaming model, events arrive at high speed, and algorithms must process them in one pass under very strict constraints of space and time. Furthermore, often the events volume is so high that it cannot be stored on disk or sent over slow network links before being processed. Instead, a streaming application can analyse continuous event streams immediately, reducing large-volume input streams to low-volume output streams for further storage, communication, or action.

The challenge is to setup a processing infrastructure able to collect information and analyse incoming event streams continuously and in real-time. Several solutions can be used in that sense. For instance, stream database systems were a very popular research topic a few years ago. Their commercial counterparts (such as Streambase4[1] or Truviso5[2]) allow users to pose queries using declarative lan-

[1] http://www.streambase.com

guages derived from SQL on continuous event streams. While extremely efficient, the functionalities of such systems are intrinsically limited by built in operators provided by the system. Another class of systems relevant to Big Data velocity are distributed stream processing frameworks. These frameworks typically propose a general-purpose, distributed, and scalable platform that allows programmers to develop arbitrary applications for processing continuous and unbounded event streams. IBM InfoSphere, StreamBase [1], Apache S4 [2], Storm [3], SAMOA and Twitter Storm are popular examples of such frameworks.

Streaming algorithms use probabilistic data structures and give fast, approximated answers. However, sequential online algorithms are limited by the memory and bandwidth of a single machine. Achieving results faster and scaling to larger event streams requires parallel and distributed computing.

The streaming paradigm is necessary to deal with the data velocity; and distributed and parallel computing to deal with the volume of data. Much recent work has attempted to address parallelism by coping data structures used for composing streams with physical architectures (e.g., clusters). This makes it easier to exploit the nested levels of hardware parallelism, which is important for handling massive data streams or performing sophisticated online analytics. Data models promoted by the NoSQL trend is addressing variety and also processing efficiency on clusters [4].

There are two approaches for dealing with streams consumption and analytics. The first one, event histories querying, supposes that there are histories or logs that are continuously fed with incoming events and that it is possible to perform dynamic and continuous (i.e., recurrent) querying and processing. The second one, complex event processing (CEP) [5], supposes that streams cannot be stored and that on-line processing and delivery are performed at given rates eventually combining them with stored data. The following sections describe these approaches.

2.2 Querying event histories

Events can be stored in event histories or logs. An event history is a finite set of events ordered by their occurrence time, and in which no two events have the same identifier. Because the number of produced events can reach thousands of events per second or higher [6], the size of an event history can be huge, increasing the difficulty of its analysis for composing events.

---

[2] http://www.dbms2.com/category/products-and-vendors/truviso/

Distributed event processing approaches, deal with events with respect to subscriptions managed as continuous queries, where results can also be used for further event compositions. According to the type of event-processing strategy (i.e., aggregation, mining, pattern look up or discovery), event-processing results can be notified as streams or as discrete results. In both cases, event processing is done with respect to events stemming from distributed producers. Provided that approaches enable dynamic and post-mortem event processing, they use different and distributed event histories for detecting event patterns. For example, the overall load of a cluster system is given by memory and CPU consumption. So, in order to compute the load model of the cluster, the event histories representing memory and CPU consumption of each computer in the cluster have to be combined and integrated with on-line event streams. Thus, histories must be analysed and correlated with on-line event streams to obtain the load (memory and CPU consumption) of the cluster.

Furthermore, event processing must handle complex subscriptions that integrate stream processing and database lookup to retrieve additional information. In order to do such kind of event processing, a number of significant challenges must be addressed. Despite the increasingly sizes of event histories, event processing needs to be fast. Filtering, pattern matching, correlation and aggregation must all be performed with low latency. The challenge is to design and implement event services that implement event processing by querying distributed histories ensuring scalability and low latency.

Continuous query processing have attracted much interest in the database community, e.g., trigger and production rules processing, data monitoring [7], stream processing [8], and publish/subscribe systems [9–11]. In contrast to traditional query systems, where each query runs once against a snapshot of the database, continuous query systems support queries that continuously generate new results (or changes to results) as new data continue to arrive [12]. Important projects and systems address continuous query processing and data streams querying. For example, OpenCQ [9], NiagaraCQ [13], Alert [14], STREAM (STanford stream datA Management) [1], Mobi-Dic [15], PLACE (Pervasive Location-Aware Computing Environments) [16, 17] and PLASTIC – IST FP6 STREP.

Concerning query languages, most proposals define extensions to SQL with aggregation and temporal operators. Languages have been proposed for expressing the patterns that applications need to observe within streams: ESPER [18], FTL, and Streams Processing Language (SPL) [19]. For example, SPL is the programming language for IBM InfoSphere Streams [20], a platform for analysing Big Data in motion meaning continuous event streams at high data-transfer rates. InfoSphere Streams processes such events with both high throughput and short response times. SPL abstracts away the complexity of the distributed system, instead exposing a simple graph-of-operators view to the user. To facilitate writing well-structured and concise applications, SPL provides higher-order composite

operators that modularize stream sub-graphs. Optimization has been addressed with respect to the characteristics of sensors [21]. Other approaches such as [22] focus on the optimization of operators. For example, to enable static checking while exposing optimization opportunities, SPL provides a strong type system and user-defined operator models.

2.3 Complex event processing

A special case of stream processing is complex event processing (CEP) [5]. CEP refers to data items in input streams as raw events and to data items in output streams as composite (or derived) events. A CEP system uses patterns to inspect sequences of raw events and then generates a composite event for each match, for example, when a stock price first peaks and then dips below a threshold. Prominent CEP systems include NiagaraCQ [23], SASE (Stream-based and Shared Event processing) [17], Cayuga [16], IBM WebSphere* Operational Decision Management (WODM) [24], Progress Apama [12], and TIBCO Business Events [25].

The challenge of CEP [5] is that there are several event instances that can satisfy a composite event type. *Event consumption* has been used to decide which component events or an event stream are considered for the composition of a composite event, and how the event parameters of the composite event are computed from its components. The *event consumption modes* are classified in recent, chronicle, continuous and cumulative event contexts (an adaptation of the parameter contexts [26, 27]).

Consider the composite event type $E_3 = (E_1 ; E_2)$ where E_1, E_2 represent event types and "$;$" denotes the operator sequence. The expression means that we are looking for patterns represented by E_3 where instances of E_1 are produced after instances of E_2. Consider the event history $H = \{\{e_{11}\}, \{e_{12}\}, \{e_{13}\}, \{e_{21}\}\}$. Thus, the event consumption mode determines which instances e_1-event(s) to combine with e_{21} for the production of instances of the composite event of type E_3. An instance of the type E_1 will be the initiator of the composite event occurrence, while an instance of type E_2 will be its terminator.

- **Recent**: Only the newest instance of the event type E_1 is used as initiator for composing an event of type E_3. In the above example, the instance e_{11} of event type E_1 is the initiator of the composite event type $E_3 = (E_1 ; E_2)$. If a new instance of type E_1 is detected (e.g. e_{12}), the older instance in the history is overwritten by the newer instance. Then, the instance e_{21} of type E_2 is combined with the newest event occurrence available: (e_{13}, e_{21}).

An initiator will continue to initiate new composite event occurrences until a new initiator occurs. When the composite event has been detected, all components

of that event (that cannot be future initiators) are deleted from the event history. Recent consumption mode is useful, e.g. in applications where events are happening at a fast rate and multiple occurrences of the same event only refine the previous value.

- **Chronicle**: For a composite event occurrence, the (initiator, terminator) pair is unique. The oldest initiator and the oldest terminator are coupled to form the composite event. In the example, the instance e_{21} is combined with the oldest event occurrence of type E_1 available: (e_{11}, e_{21}).

In this context, the initiator can take part in more than one event occurrence, but the terminator does not take part in more than one composite event occurrence. Once the composite event is produced, all constituents of the composite event are deleted from the event history. The chronicle consumption mode is useful, e.g. in application where there is a connection between different types of events and their occurrences, and this connection needs to be maintained.

- **Continuous**: Each initiator event starts the production of that composite event. The terminator event occurrence may then trigger the production of one or more occurrences of the same composite event, i.e. the terminator terminates those composite events where all the components have been detected (except for the terminator). In the example, e_{21} is combined with all event of type E_1: (e_{11}, e_{21}), (e_{12}, e_{21}) and (e_{13}, e_{21}); and does not delete the consumed events.

The difference between the continuous and the recent and chronicle consumption modes is that in the latter one initiator is coupled with one terminator, whereas the continuous consumption mode one terminator is coupled with one or many initiators. In addition, it adds more overhead to the system and requires more storage capacity. This mode can be used in applications where event detection along a moving time window is needed.

- **Cumulative**: All occurrences of an event type are accumulated until the composite event is detected. In the example, e_{21} is combined with all event occurrences of type E_1 available $(e_{11}, e_{12}, e_{13}, e_{21})$.

When the terminator has been detected, i.e. the composite event is produced; all the event instances that constitute the composite event are deleted from the event history. Applications use this context when multiple occurrences of component events need to be grouped and used in a meaningful way when the event occurs.

2.4 Target applications

Big Data is no longer just the domain of actuaries and scientists. New technologies have made it possible for a wide range of people – including humanities and social science academics, marketers, governmental organizations, educational in-

stitutions, and motivated individuals – to produce, share, interact with, and organize data. This section presents three challenges where Big Data velocity is particularly important and it is an enabling element for addressing application requirements: digital shadow analytics that relates velocity, volume, and value; smart cities and urban computing and industry 4.0 that relate velocity and volume and veracity.

2.4.1 Extracting value out of the digital shadow

The digital shadow of individuals is growing faster every year, and most of the time without knowing it. Our digital shadow is made up of information we may deem public but also data that we would prefer to remain private. Yet, it is within this growing mist of data where big data opportunities lie — to help drive more personalized services, manage connectivity more efficiently, or create new businesses based on valuable, yet-to-be-discovered intersections of data among groups or masses of people.

Today, social-network research involves mining huge digital data sets of collective behaviour online. The convergence of these developments — mobile computing, cloud computing, big data, and advanced data mining technologies — is compelling many organizations to transition from a "chasing compliance" mind set to a risk management mind set. Big streams' value comes from the patterns that can be derived by making connections between pieces of data, about an individual, about individuals in relation to others, about groups of people, or simply about the structure of information itself.

Advertisers, for instance, would originally publicize their latest campaigns statically using pre-selected hash-tags on Twitter. Today, real-time data processing opens the door to continuous tracking of their campaign on the social networks, and to online adaptation of the content being published to better interact with the public, e.g., by augmenting or linking the original content to new content, or by reposting the material using new hash-tags. Online social networks, like Facebook or Twitter, are increasingly responsible for a significant portion of the digital content produced today. As a consequence, it becomes essential for publishers, stakeholders and observers to understand and analyse the data streams originating from those networks in real-time. However, current (de-facto standard) solutions for big data analytics are not designed to deal with evolving streams.

Open issues are related with the possibility of processing event streams (volume) in real-time (velocity) in order to have a continuous and accurate views of the evolution of the digital shadow (veracity). This implies to make event streams processing scale and provide support for making decisions on which event histories should persist and those that are volatile, those that should be filtered and correlated to have different perspectives of peoples and crowds digital shadow ac-

cording to application requirements. Event stream types, processing operators, adapted algorithms and infrastructures need to be understood and revisited for addressing digital shadow related challenges.

2.4.2 Smart cities and urban computing

The development of digital technologies in the different disciplines, in which cities operate, either directly or indirectly, is going to alter expectations among those in charge of the local administration. Every city is a complex ecosystem with a lot of subsystems to make it work such as work, food, cloths, residence, offices, entertainment, transport, water, energy etc. With the growth there is more chaos and most decisions are politicised, there are no common standards and data is overwhelming.

Smart cities are related to sensing the city's status and acting in new intelligent ways at different levels: people, government, cars, transport, communications, energy, buildings, neighbourhoods, resource storage, etc. A vision of the city of the "future", or even the city of the present, remains on the integration of science and technology through information systems. For example, unlike traditional maps, which are often static representations of distributed phenomena at a given moment in time, Big data collection tools can be used for grasping the moving picture of citizens' expressions, as they are constantly changing and evolving with the city itself (identifying urban areas and colour them according to the time period of the day they are pulsing the most) [28].

Big data streams can enable online analysis of users' perceptions related to specific geographic areas; and post-mortem analysis for understanding how specific user groups use public spaces; discover meaningful relationships and connections between places, people and uses. Big event histories can be analysed for understanding how specific features of city spaces, services and events affect people's emotions; detect post-event/fair reactions and comments by citizens and participants. These analytics processes can support the development of tools aimed at assisting institutions and large operators, involved in monitoring, designing and implementing strategies and policies oriented to improve the responsiveness of urban systems to the requests of citizens and customers.

2.4.3 Robotics and Industry 4.0

Big data analytics and cloud architectures allow leveraging large amounts of structured, unstructured and fast-moving data. Putting this technology into robotics can lead to interesting dimensions to well-known problems like SLAM and lower-skilled jobs executions (assembly line, medical procedures and piloting vehicles). Rather than viewing robots and automated machines as isolated systems,

Cloud Robotics and Automation is a new paradigm where robots and automation systems exchange data and perform computation via networks. Extending work linking robots to Internet, Cloud Robotics and Automation builds an emerging research in cloud computing, machine learning, big data, and industry initiatives in the Internet of Things, Industrial Internet, and Industry 4.0.

For example, SLAM is a technique used by digital machines to construct a map of an unknown environment while keeping track of the machine's location in the physical environment. This requires a great deal of computational power to sense a sizable area and process the resulting data to both map and localize. Complete 3D SLAM solutions are highly computationally intensive as they use complex real-time particle filters, sub-mapping strategies or combination of metric topological representations. Robots using embedded systems cannot fully implement SLAM because of their limitation in computing power. Big Data can enable interactive data analysis with real-time answers that can empower intelligent robots to analyse enormous and unstructured datasets (big data analytics) to perform jobs. This of course requires the processing of huge amounts of event streams coming from robots that must be processed efficiently to support on-line dynamic decision-making.

3 Event stream models

A vast number of event management models and systems have been and continue to be proposed. Several standardization efforts are being made to specify how entities can export the structure and data transported by events. Existing models have been defined in an *ad hoc* way, notably linked to the application context (active DBMS event models), or in a very general way in middleware (Java event service, MOMs). Of course, customizing solutions prevents systems to be affected with the overhead of an event model way too sophisticated for their needs. However, they are not adapted when the systems evolve, cooperate and scale, leading to a lack of adaptability and flexibility.

This section introduces the background concepts related to event streams and related operators. It mainly explains how event types become streams and how this is represented in models that are then specialized in concrete event stream systems.

3.1 From event types to streams

The literature proposes different definitions of an event. For example, in [29] an event is a happening of interest, which occurs instantaneously at a specific

time. [30] characterizes an event as the instantaneous effect of the termination of an invocation of an operation on an object. In this document we define an event in terms of a source named *producer* in which the event occurs, and a *consumer* for which the event is significant.

An event type characterizes a class of significant facts (events) and the context under which they occur. An event model gives concepts and general structures used to represent event types. According to the complexity of the event model, the event types are represented as sequences of strings [31], regular expressions – patterns – [32] or as expressions of an event algebra [26, 33, 34]. In certain models, the type itself contains implicitly the contents of the message. Other models represent an event type as a collection of *parameters* or *attributes*. For example, UpdateAccount`(idAccount:string, amount:real)` is an event type that represents the updates executed on an account with number `idAccount` and where the amount implied in the operation is represented by the attribute `amount`. `Event` types have at least two associated parameters: an *identifier* and a *timestamp*.

In addition, an event type can have other parameters describing the circumstances in which events occurred. This information describes the ***event production environment*** or ***event context***. In some models, the event type is represented by a set of tuples of the form (*variable*, *domain*). Generally, these parameters represent for instance the agents, resources, and data associated with an event type, the results of the action (e.g., return value of a method), and any other information that characterizes a specific occurrence of that event type. For example, in active systems, the parameters of an event are used to evaluate the condition and to execute the action of an ECA rule.

Event types can be classified as *primitive event types* that describe elementary facts, and *composite event types* that describe complex situations by event combinations.

A **primitive event type** characterizes an atomic operation (i.e., it completely occurs or not). For example, the update operation of an attribute value within a structure, the creation of a process. In the context of databases, primitive event types represent data modification (e.g. the insertion, deletion or modification of tuples), transactions processing (e.g. begin, commit or abort transactions). In an object-oriented context, a method execution can be represented by a primitive event type.

Many event models classify the primitive event types according to the type of operations they represent (databases, transactional, applicative). These operations can be classified as follows:

- ***Operations executed on data:*** an operation executed on a structure, for example, a relational table, an object. In relational systems this operation can correspond to an insert/update/delete operation applied to one or more n-tuples. In object-based systems, it can be a read/write operation of an object attribute.

- ***Operations concerning the execution state of a process:*** events can represent specific points of an execution. In DBMS, events can represent execution points of a transaction (before or after the transaction delete/commit). In a workflow application, an event can represent the beginning (end) of a task. The production of exceptions within a process can be represented by events.
- ***User operations:*** an operation on a widget in an interactive interface, the connection of the user to the network, correspond to events produced by a user.
- ***Operations produced within the execution context:*** events can represent situations produced in the environment: (i) specific points in time (clock), for example, it is 19:00 or 4 hours after the production of an event; (ii) events concerning to the operating system, the network, etc.

A **composite event type** characterizes complex situations. A composite event type can be specified as a regular expression (often called a *pattern*) or as a set of primitive or other composite event types related by event algebra operators such as disjunction, conjunction, sequence). For example, consider the composite event type represented as the regular expression $(E_1 \mid E_2)^* E_3$ where E_1, E_2, and E_3 are event types, "|" represents *alternation*[3], and "*" represents the *Kleene closure*[4]. The composite event type $E_4 = E_1 \ op \ (E_2 \ op \ E_3)$ is specified by an event algebra where E_1, E_2, and E_3 are primitive or composite event types, and op can be any binary composition operator, e.g. disjunction, conjunction, sequence.

- The ***occurrence*** or ***instance*** of an event type is called an event. Events occur in time and then they are associated to a point in time called ***event occurrence time*** or ***occurrence instant***. The occurrence time of an event is represented by its timestamp. The timestamp is an approximation of the event occurrence time. The accuracy of timestamps depends on the event detection strategy and on the *timestamping* method.

The granularity used for representing time (day, hour, minute, second, etc.) is determined by the system. Most event models and systems assume that the *timeline* representation corresponds to the Gregorian calendar time, and that it is possible to transform this representation as an element of the discrete time domain having 0 (zero) as origin and ∞ as limit. Then, a point in time (event occurrence time) belongs to this domain and it is represented by a positive integer. The event $(\texttt{updateAccount(idAccount:0024680, amount:24500)}, t_i)$ is an occurrence of the event type `UpdateAccount(idAccount:string, amount:real)` produced at time t_i, where the time may represent an instant, a duration or an interval. A *duration* is a period of time with known length, e.g. 8

---

[3] $E_1 \mid E_2$ matches either events of type E_1 or E_2.

[4] E^* is the concatenation of zero or more events of type E.

seconds. An *interval* is specified by two instants as `[01/12/2006, 01/12/2007]`.

The notion of event type provides a static view of a happening of interest or a behaviour pattern in time. Yet it is not sufficient to represent the dynamic aspect of events flowing, that is to say, being produced at a continuous rate. The notion of stream provides means to represent event flows.

3.2 Event streams

An event stream is represented by an append-only sequence of events having the same type T. We note *Stream*(*T*) the stream of events of type T. Event streams can be continuous, or potentially unbounded (i.e. events can be inserted in a stream at any time). A finite part of an event stream of type T is noted *Stream$_f$(T)*. In order to define how to deal with this "dynamic" structure used to consume a continuous flow of event occurrences, several works have proposed operators to represent the partition of the stream so that this partitions can be processed and consumed in continuous processes [35].

The operator `window` is the most popular one. A `window` partitions an event stream into finite event streams. The result is a stream of finite streams, which we note `Stream(Streamf(E))`. The way each finite stream is constructed depends on the window specification, which can be time-based or tuple-based.

3.2.1 Time based windows

Time based windows define windows using time intervals.

- *Fixed window*: `win:within(`t_b`,` t_e`,` ES_i`)`. Defines a fixed time interval `[tb, te]`. The output stream contains a single finite event stream E_iS_{jf} such that an event e_i of type E_i belong to ES_{if} iff $t_b \leq e_i$`.receptionTime` $\leq t_e$.
- *Landmark window*: `win:since(`t_b`,` ES_i`)`. Defines a fixed lower bound time t_b. The output stream is a sequence of finite event streams $ES_{i,kf}$ `k = 1,…n` such that each $ES_{i,kf}$ contains events e_i received since the time lower bound t_b. That is, $\forall\, k,\ e_i \in ES_{i,kf}$ iff $t_b \leq e_i.receptionTime$.
- *Sliding window*: `win:sliding(`t_w`,` t_s`,` ES_i`)`. Defines a time duration t_w and a time span t_s. The output stream is a sequence of finite event streams $ES_{i,kf}$ `k = 1,…n` such that each $ES_{i,kf}$ contains events of type E_i produced during t_w time unit. The finite event streams in the sequence are produced each `ts` time unit. That is, if $ES_{i,kf}$ is produced at time `t`, then $ES_{i,k+1}$ will be produced at time `t+`t_s.

3.2.2 Tuple based windows

Tuple based windows define the number of events for each window.

- *Fixed size windows*: `win:batch(`n_b`, ESi)`. Specifies a fixed size n_b of each finite stream. The output stream is a sequence of finite event streams $ES_{i,kf}$ `k = 1,…n,` each finite event stream $ES_{i,kf}$ containing n_b most recent events and are non-overlapping. If we consider windows of size 3, the event stream ES_i, = {$e_{i,1}$, $e_{i,2}$, $e_{i,3}$, $e_{i,4}$, $e_{i,5}$, $e_{i,6}$, …} will be partitioned in finite event streams {$ES_{i,1f}$, $ES_{i,2f}$, . . .} such that $ES_{i,1f}$ = { $e_{i,1}$, $e_{i,2}$, $e_{i,3}$}, $ES_{i,2f}$ = { $e_{i,4}$, $e_{i,5}$, $e_{i,6}$}, and so on.
- *Moving fixed size windows*: `win:mbatch (nb, m,` ES_i`)`. Defines a fixed size n_b of each finite stream, and a number of events `m` after which the window moves. The output stream is a sequence of finite event streams $ES_{i,kf}$ `k = 1,…n` such that each $ES_{i,kf}$ contains n_b most recent events of type E_i,. $ES_{i,k+1}$ is started after `m` events are received in $ES_{i,kf}$ (moving windows). As result, an event instance may be part of many finite event streams. This is the case if `m` ≤ `nb`. For example, if we consider windows of size `nb=3` moving after each `m = 2` events, the event stream `ESi = {ei,1, ei,2, ei,3, ei,4, ei,5, ei,6, ei,7, …}` will be partitioned into finite event streams `{ESi,1f, ESi,2f, ESi,3f , …}` such that `ESi,1f ={ei,1, ei,2, ei,3}`, `ESi,2f = {ei,3, ei,4, ei,5}`, `ESi,3f = { ei,5, ei,6, ei,7}`, and so on.

The notions of type and event occurrence are useful for dealing with event streams processing phases. Event types are important when addressing the expression of interesting happenings that can be detected and observed within a dynamic environment. As discussed in this section, it is possible to associate to the notion of type, operators that can be applied for defining complex event types. An event definition language can be developed using such operators. The notion of event occurrence is useful to understand and model the association event – time, and then model a continuous flow under the notion of stream. Then it is possible to define strategies for composing streams (on-line) and for analysing streams. These strategies are discussed in the following section.

4 Complex event composition

Event composition is the process of producing composite events from detected (primitive and composite) event streams. Having a composite event implies that there exists a relation among its component events, such causal order and temporal relationships.

Several academic research and industrial systems have tackled the problem of event composition. Techniques such as complex pattern detection [33, 34, 36, 37], event correlation [38], event aggregation [6], event mining [39, 40] and stream processing [41–43], have been used for composing events. In some cases event composition is done on event histories (e.g. event mining) and in other cases it is done dynamically as events are produced (e.g. event aggregation and stream processing). Analysing an event history searching or discovering patterns produces events. Composite events can be specified based on an *event composition algebra* or *event patterns*.

This section introduces different strategies used for composing event streams. In general these strategies assume the existence of a history (total or partial) and thus adopt monotonic operators, in the case of algebras, or monotonic reasoning in the case of chronicles or codebooks and rules used as composition strategies. Rule based approaches assume that events streams patterns can be detected and they can trigger rules used to notify them.

4.1 Composition algebras

An algebra defines a collection of elements and operators that can be applied on them. Thus, the event composition algebra defines *operators* for specifying composite event types based on primitive or composite event types related by event operators. An event composition algebra expression is of the form E_1 op E_2. The construction of such an expression produces composite events. The types of these events depend on the operators. Operators determine the order in which the component events must occur for detecting the specified composite event. The occurrence time and parameters of a composite event depend on the semantics of the operator. Therefore, the parameters of a composite event are derived from the parameters of its component events depending on the operator.

Many event models that characterize composite events consider operators such as disjunction, conjunction and sequence. Others add the selection and negation operators. In the following paragraphs, we classify the event operators in: *binary*, *selection* and *temporal operators*. Table 1 synthesizes the operator families that can be used for composing event streams. Their definition is presented in Appendix A.

FILTERING

Window

Time based

	Fixed	win:within(t_b, t_e, ES_i)
	Landmark	win:since(tb, ESi)
	Sliding	win:sliding(tw, ts, ESi)

Tuple based

	Fixed size	win:batch(nb, ESi)
	Moving fixed size	win:mbatch (nb, m, ESi)

Selection

	First occurrence	(*E in H)
	History	(Times(n, E) in H)
	Negation	(Not E in H)

BINARY (temporal correlation)

Instance based

	Disjunction	(E_1 | E_2)
	Conjunction	(E_1 , E_2)
	Sequence	(E_1 ; E_2)
	Concurrency	(E_1 ‖ E_2)

Interval based

	During	(E_2 during E_1)
	Ovelap	(E_1 overlaps E_2)
	Meet	(E_1 meets E_2)
	Start	(E_1 starts E_2)
	End	(E_1 ends E_2)

TEMPORAL

	Temporal offset
	Interval expresions

Table 1 Event composition operators

The operators are used for defining well-formed algebraic expressions considering their associativity and commutability properties. Combined the estimated execution cost it is possible to propose optimization strategies for reducing event stream production.

4.2 Composition techniques

Different techniques can be adopted for composing event streams, depending whether this task is done on-line or post-mortem. For an on-line composition, automata (of different types) are the most frequent adopted structure. When detected event streams are continuously (i.e., recurrently) fed to nodes that process them and disseminate them to other nodes that implement composition operators. Depending on the type of automaton, different patterns can be produced and delivered to consumers. Post-mortem event streams composition, assume that it is possible to store all events streams produced during a given period, or at least a representative sample of event streams. These event histories sometimes called event traces are used to apply knowledge discovery techniques seeking to extract patterns, correlations, to understand and predict behaviour models. The following sections introduce prominent examples of these techniques.

4.2.1 Automata oriented event composition

In current research projects, the composition process is based on the evaluation of abstractions such as *finite state automata*, *Petri nets*, *matching trees* or *graphs*.

- **Finite state automata**: Considering that composite event expressions are equivalent to regular expressions if they are not parameterized, it is possible to implement them using finite state automata. A first approach using automata has been made in the active data base system Ode [36, 44, 45]. An automaton can be defined for each event, which reaches an accepting state exactly whenever the event occurs. The event history provides the sequence of input events to the automaton. The event occurrences are fed into the automaton one at a time, in the order of their event identifiers. The current marking of an automaton determines the current stage of the composition process. If the automaton reaches an accepting state, then the composite event implemented by the automaton occurs. Nevertheless, automata are not sufficient in case of event parameters have to be supported. The automata have to be extended with a data structure that stores the event parameters of the primitive events from the time of their occurrence to the time at which the composite event is detected.
- **Petri nets** are used to support the detection of composite events that are composed of parameterized events. SAMOS [33, 46] uses the concepts of Coloured Petri nets and modifies them to so-called SAMOS Petri Nets. A Petri net consists of places, transitions and arcs. Arcs connect places with transitions and transitions with places. The places of a Petri net correspond to the potential states of the net, and such states may be changed by the transitions. Transitions correspond to the possible events that may occur (perhaps concurrently). In Coloured Petri nets, tokens are of specific token types and may carry complex information. When an event occurs, a corresponding token is inserted into all places representing its event type. The flow of tokens through the net is then determined; a transition can fire if all its input places contain at least one token. Firing a transition means removing one token from each input place and inserting one token into each output place. The parameters corresponding to the token type of the output place are derived at that time. Certain output places are marked as end places, representing composite events. Inserting a token into an end place corresponds to the detection of a composite event.
- **Trees**: Another approach to implement event composition uses matching trees that are constructed from the composite event types. The leaves represent primitive event types. The parent nodes in the tree hierarchy represent composite event types. Primitive events occur and are injected into the leaves corresponding to their event type. The leaves pass the primitive events directly to their parent nodes. Thus, parent nodes maintain information for matched events, such as mapping of event variables and matching event instances. A composite event is detected if the root node is reached and the respective event data are successfully filtered.
- **Graph-based** event composition has been implemented by several active rule systems like SAMOS [26, 27, 47], Sentinel [48] and NAOS [34]. An event graph, is a Direct Acyclic Graph (DAG) that consists of non-terminal nodes (N-nodes), terminal nodes (T-nodes) and edges [26]. Each node represents either a primitive event or a composite event. N-nodes represent composite events and

may have several incoming and several outgoing edges. T-nodes represent primitive events and have one incoming and possibly several outgoing edges. When a primitive event occurs, it activates the terminal node that represents the event. The node in turn activates all nodes attached to it via outgoing edges. Parameters are propagated to the nodes using the edges. When a node is activated, the incoming data is evaluated (using the operator semantics of that node and the consumption mode) and if necessary, nodes connected to it are activated by propagating the parameters of the event. If the node is marker as a final node, the corresponding composite event is signalled.

These structures are well adapted for on-line stream event composition where windows and filters are used for controlling the consumption rate of streams combined with other processing operators for causally or temporally correlating them. These structures can also be matched towards parallel programs that can make in some cases event stream composition more efficient. Having parallel programs associated to these composition structures has not yet been widely explored. The emergence of the map-reduce and data flow model and associated infrastructures can encourage the development of solutions adapted for addressing Big Data velocity and volume.

4.2.2 Event correlation

The process of analysing events to infer a new event from a set of related events is defined as *event correlation*. It is mostly used to determine the root cause of faults in network systems [49]. Thus, an *event correlation system* correlates events and detects composite events. There are several methods for correlating events, including *compression*, *count*, *suppression*, and *generalization*. **Compression** reduces multiple occurrences of the same event into a single event, allowing to see that an event is recurring without having to see every instance individually. **Count** is the substitution of a specified number of similar events (not necessarily the same event) with a single event. **Suppression** associates priorities with events, and may hide a lower priority event if a higher priority event exists. Finally, in **generalization** the events are associated with a superclass that is reported rather than the specific event.

Other methods of event correlation are by ***causal*** relationships (i.e., event *A* causes event *B*), and by ***temporal*** correlations where there is a time period associated with each possible correlation, and if the proper events occur during a particular time period, they may be correlated. Event correlation techniques have been derived from a selection of computer science paradigms (AI, graph theory, information theory, automata theory) including *rule-based systems*, *model based reasoning systems*, *model traversing techniques*, *code-based systems*, *fault propagation models* and the *code-book approach*.

Rule-based systems [50, 51] are composed of rules of the form **if** condition **then** conclusion. The condition part is a logical combination of propositions about the current set of received events and the system state; the conclusion determines the state of correlation process. For example, a simple rule that correlates the event occurrences e_1 of type E_1 and e_2 of type E_2 for producing an event e_3 is: **if** e_1 and e_2 **then** e_3. The system operation is controlled by an inference engine, which typically uses a forward-chaining inference mechanism.

In [49] composite events are used for event correlation. It presents a composite event specification approach that can precisely express complex timing constraints among correlated event instances. A composite event occurs whenever certain conditions on the attribute values of other event instances become true, and is defined in the following format:

```
define composite event CE with
attributes ([NAME, TYPE], …, [NAME, TYPE])
which occurs
whenever timing condition
 TC is [satisfied | violated]
if condition
 C is true
then
 ASSIGN VALUES TO CE's ATTRIBUTES;
```

The rules for correlation reflect the relationship among the correlated events, such as causal or temporal relationship. If these relationships can be specified in the composite event definitions, the results of correlation are viewed as occurrences of the corresponding composite events. Thus, relationships among events for correlation, either causal or complex temporal are expressed as conditions on event attributes for composite events. Considering a common event correlation rule in networks with timing requirements: "when a link-down event is received, if the next link-up event for the same link is not received within 2 minutes and an alert message has not been generated in the past 5 minutes, then alert the network administrator"; the composite event LinkADownAlert is defined as follows:

```
define composite event LinkADownAlert with
attributes (["Occurrence Time" : time]
            ["Link Down Time" : time])
which occurs
whenever timing condition
not LinkUp in [occTime(LinkADown),
               occTime(LinkADown)+2min]
and not LinkADownAlert in
        [occTime(LinkADown)-3min,
         occTime(LinkADown)+2min]
is satisfied
```

```
if condition true is true
then {
"Link Down Time" := occTime(LinkADown);
}
```

where `LinkADown` and `LinkAUp` correspond to the up and down events of a link `A`. The composite event will occur at 2 minutes after an occurrence of `LinkADown` event if no `LinkAUp` event occurs during 2-minute interval and no `LinkADownAlert` event was triggered during the past 5-minute interval.

Hence, since the composite events are used to represent the correlation rules, the correlation process is essentially the task of composite event detection through event monitoring. Therefore, if some time constraints are detected as being satisfied or violated according to the composite event definitions, the condition evaluator is triggered. The conditions on other event attributes are evaluated. Once the conditions are evaluated as true, the attribute values of the corresponding composite event are computed and their occurrences are triggered. As a result of the hard-coded system connectivity information within the rules, rule-based systems are believed to lack scalability, to be difficult to maintain, and to have difficult to predict outcomes due to unforeseen rule interactions.

Model-based reasoning incorporates an explicit model representing the ***structure*** (static knowledge) and ***behavior*** (dynamic knowledge) of the system. Thus, the model describes dependencies between the system components and/or causal relationships between events. Model-based reasoning systems [49, 52, 53] utilize inference engines controlled by a set of correlation rules, whose conditions usually contain model exploration predicates. The predicates test the existence of a relationship among system components. The model is usually defined using an object-oriented paradigm and frequently has the form of a graph of dependencies among system components.

In the **codebook** technique [54] causality is described by a causality graph whose nodes represent events and whose directed edges represent causality. Nodes of a causality graph may be marked as problems (`P`) or symptoms (`S`). The causality graph may include information that does not contribute to correlation analysis. For example a cycle represents causal equivalence. Thus, a cycle of events can be aggregated into a single event. Similarly, certain symptoms are not directly caused by any problem but only by other symptoms. They do not contribute any information about problems that is not already provided by these other symptoms. These indirect symptoms may be eliminated without loss of information. The information contained in the correlation graph must be converted into a set of codes, one for each problem in the correlation graph. A code is simply a vector of `0`s and `1`s. The value of `1` at the i^{th} position of a code generated for problem p_j indicates cause-effect implication between problem p_j and symptom s_i. The *codebook* is a subset of symptoms that has been optimized to minimize the number of symptoms

that have to be analysed while ensuring that the symptom patterns distinguish different problems.

4.2.3 Chronicle recognition

A *chronicle* is a set of events, linked together by time constraints [DGG93, Gha96, Dou96]. The representation of chronicles relies on a propositional reified logic formalism where a set of multi-valued domain attributes are temporally qualified by predicates such as `event` and `hold`.

The persistence of the value `v` of a domain attribute `p` during the interval `[t,t']` is expressed by the assertion:

```
hold(p:v,(t,t')).
```

An event is a change of the value of a domain attribute, the predicate `event` is defined through the predicate hold:

```
event(p:(v1,v2),t) ≡
∃τ<t<τ'|hold(p:v1,(τ,t))^hold(p:v2,(t,τ'))^(v1≠v2)
```

A *chronicle model* represents a piece of the evolution of the world; it is composed of four parts: (i) a set of events which represents the relevant changes of he world for this chronicle; (ii) a set of assertions which is the context of the occurrences of the chronicle events; (iii) a set of temporal constraints which relates events and assertions between them; and (iv) a set of actions which will be processed when the chronicle is recognized.

The chronicle recognition is complete as long as the observed event stream is complete. This hypothesis enables to manage context assertions quite naturally through occurrences and non-occurrences of events. Then, to process assertion `hold(p:v,(t,t'))`, the process verifies that there has been an `event(p:(v',v),t")` with `t"<t` and such that no event `p:(v,v")` occurs within `[t",t']`.

The recognition process relies on a complete forecasting of expected events predicted by chronicle. An interval, called *window of relevance* `D(e)` is defined, which contains all possible occurrence times for a predicted event `e` of a partial instance `S`, in order for `e` to be consistent with constraints and known times of observed events in `S`.

A chronicle instance may progress in two ways: (i) a new event may be detected, it can be either integrated into the instance and make the remaining predictions more precise, or it may violate a constraint for an assertion and make the corresponding chronicle invalid; or (ii) time passes without nothing happening and, perhaps, may make some deadline violated or some assertions constraints obsolete.

When an observed event e matches an model event e_k, the reception time $r(e)=now$, and either

- $d(e) \notin D(e_k)$: e does not meet the temporal constraints of the expected event e_k of S,
- $d(e) \in D(e_k)$: $D(e_k)$ is reduced to the single point $d(e)$.

The reduction must be propagated to other expected events, which in turn are further constrained; i.e., temporal windows are updated.

```
propagate(e_k, S)
     for all forthcoming event e_i ≠ e_k of S
            D(e_i) ← D(e_i) ∩ [D(e_k) + I(e_i - e_k)]
```

This produces a new set of non-empty and consistent $D(e_i)$. In addition, when the internal clock is updated, this new value of now can reduce some windows of relevance $D(e_i)$ and, in this case, it is needed to propagate it over all expected events of an instance S: $D(e_i) \leftarrow D(e_i) \cap ([t, +\infty] - D(ei))$. A clock update does not always require propagation, it is necessary to propagate only when a time bound is reached. Therefore, time bounds enable an efficient pruning.

When an event is integrated in a chronicle, the original chronicle instance must be duplicated before the temporal window propagation, and only the copy is updated. For each chronicle model, the system manages a tree of current instances. When a chronicle instance is competed or killed, it is removed from this tree. Duplication is needed to warranty the recognition of a chronicle as often as it may arise, even if its instances are temporally overlapping.

The size of the tree hypotheses is the main source of complexity. Using duration thresholds in a chronicle model is a strategy to reduce its size. Further knowledge restricting multiple instances of events is also beneficial. There may also be chronicles that cannot have two complete instances that overlap in time or share a common event; the user may also be interested in recognizing just one instance at a time. For both cases, when a chronicle instance is recognized, all its pending instances must be removed.

4.2.4 Event and traces mining

Data mining, also known as Knowledge-Discovery in databases (KDD) is the practice of automatically analysing large stores of data for patterns and then summarizing them as useful information. Data mining is sometimes defined as the process of navigating through the data and trying to find out patterns and finally establishing all relevant relationships. Consequently, the event-mining goal is to identify patterns that potentially indicate the production of an event within large event data sets. Event mining adopts data mining techniques for the recognition of

event patterns, such as association, classification, clustering, forecasting, etc. Therefore, events within a history can be mined in a multitude of ways: unwanted events are filtered out, patterns of logically corresponding events are aggregated into one new composite event, repetitive events are counted and aggregated into a new primitive event with a count of how often the original event occurred, etc.

Event correlation approaches may be further classified as state-based or stateless. Stateless systems typically are only able to correlate alarms that occur in a certain time-window. State-based systems support the correlation of events in an event-driven fashion at the expense of the additional overhead associated with maintaining the system state.

4.3 Discussion

This section presented expressions of an algebra for composing events. It gave a classification of algebraic operators that can be defined depending on whether events are considered instantaneous happenings or processes with duration represented as intervals.

Event composition in large-scale systems provides a means of managing the complexity of a vast number of events. Large-scale event systems need to support event composition in order to quickly and efficiently notify relevant complex information. In addition, distributed event composition can improve efficiency and robustness of systems. Thus, event types can be related and thus denote a new complex event type. Relationships between event types can be expressed by an event composition algebra.

The different event-based approaches are characterized by their means for specifying and detecting primitive and composite events. The composition process is based on the evaluation of abstractions such as finite state automata, Petri nets, matching trees, graphs. While event tracing enables the detection of performance problems at a high level of detail, growing trace-file size often constrains its scalability on large-scale systems and complicates management, analysis, and visualization of trace data. Such strategies can cope to Big streams velocity as long as they can be efficiently used or that they can be exploited in parallel in order to ensure good performance.

5 Conclusion and outlook

Typical Big Data analytics solutions such as batch data processing systems can scale-out gracefully and provide insight into large amounts of historical data at the

expense of a high latency. They are hence a bad fit for online and dynamic analyses on continuous streams of potentially high velocity.

Building robust and efficient tools to collect, analyse, and display large amounts of data is a very challenging task. Large memory requirements often cause a significant slow down or - even worse - place practical constraints on what can be done at all. Moreover, if when streams stem from different providers, before merging those streams into a single global one, the merge step may require a large number of resources creating a potential conflict with given infrastructure limits. Thus, the amount of (event) streams poses a problem for (i) management, (ii) visualization, and (iii) analysis. The size of a stream history may easily exceed the user or disk quota or the operating system imposed file-size limit of 2 GB common on 32-bit platforms. Very often these three aspects cannot be clearly separated because one may act as a tool to achieve the other, for example, when analysis occurs through visualization.

Even if the data management problem can be solved, the analysis itself can still be very time consuming, especially if it is performed without or with only little automatic support. On the other hand, the iterative nature of many applications causes streams to be highly redundant. To address this problem, stream collection must be coupled with efficient automatic cleaning techniques that can avoid redundancy and information loss.

Existing, event stream approaches and systems seem to be adapted for dealing with velocity but do not completely scale when volume becomes big. Efficient parallel algorithms exploiting computing resources provided by architectures like the cloud can be used to address, velocity at the different phases of Big stream cycle: collection, cleaning and analysis. The huge volume of streams, calls for intelligent storage methods that can search for a balance between volume, veracity and value. Representative stream samples must be stored to support static analytics (e.g., event trace mining) while continuous on-line composition processes deal with streams and generate a real-time vision of the environment. Concrete applications are already calling for such solutions in order to build smarter environments, social and individual behaviours, and sustainable industrial processes.

6 References

1. Terry, D., Goldberg, D., Nichols, D., Oki, B.: Continuous queries over append-only databases. ACM SIGMOD Record (1992).

2. Zheng, B., Lee, D.L.: Semantic caching in location-dependent query processing. In Proc. of SSTD (2001).

3. Urhan, T., Franklin, M.J.: Xjoin: A reactively-scheduled pipelined join operator. IEEE Data Eng. Bull. 23, (2000).

4. Adiba, M., Castrejón, J.C., Espinosa-Oviedo, J.A., Vargas-Solar, G., Zechinelli-Martini, J.L.: Big Data Management: Challenges, Approaches, Tools and their limitations. Networking for Big Data (2015).

5. Abiteboul, S., Manolescu, I., Benjelloun, O., Milo, T., Cautis, B., Preda, N.: Lazy query evaluation for active xml. In Proc. of the SIGMOD Int. Conf. on Management of Data (2004).

6. Luckham, D.: The Power of Events: An Introduction to Complex Event Processing in Distributed Systems. Addison Wesley Professional (2002).

7. Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., Zdonik, S.: Monitoring streams: a new class of data management applications. In Proc. of the 28th Int. Conf. on Very Large Data Bases (VLDB '02) (2002).

8. Babu, S., Widom, J.: Continuous queries over data streams. SIGMOD Rec. 30, (2001).

9. Liu, L., Pu, C., Tang, W.: Continual queries for internet scale event-driven information delivery. IEEE Trans. Knowl. Data Eng. 11, (1999).

10. Chen, J., DeWitt, D.J., Tian, F., Wang, Y.: NiagaraCQ: a scalable continuous query system for Internet databases. In Proc. of the 2000 ACM SIGMOD Int. Conference on Management of Data (SIGMOD '00). pp. 379–390. , New York, USA (2000).

11. Dittrich, J.-P., Fischer, P.M., Kossmann, D.: Agile: adaptive indexing for context-aware information filters. In Proc. of the SIGMOD Int. Conf. on Management of Data (2005).

12. Agarwal, P.K., Xie, J., Hai, Y.: Scalable continuous query processing by tracking hotspots. Proc. Very Large Data Bases. (2006).

13. Chen, J., DeWitt, D.J., Tian, F., Wang, Y.: Niagracq: A scalable continuous query system for internet databases. In Proc. of SIGMOD Int. Conf. on Management of Data (2000).

14. Schreier, U., Pirahesh, H., Agrawal, R., Mohan, C.: Alert: An architecture for transforming a passive dbms into an active dbms. Proc. Int. Conf. Very Large Data Bases. (1991).

15. Cao, H., Wolfson, O., Xu, B., Yin, H.: Mobi-dic: Mobile discovery of local resources in peer-to-peer wireless network. IEEE Data Eng. Bull. 28, (2005).

16. Mokbel, M.F., Xiong, X., Aref, W.G., Hambrusch, S., Prabhakar, S., Hammad, M.: Place: A query processor for handling real-time spatiotemporal data streams (demo). Proc. Very Large Data Bases. (2004).

17. Hellerstein, J.M., Franklin, M.J., Chandrasekaran, S., Deshpande, A., Hildrum, K., Madden, S., Raman, V., Shah, M.A.: Adaptive query processing: Technology in evolution. IEEE Data Eng. Bull. (2000).

18. Anicic, D., Fodor, P., Rudolph, S., Stühmer, R., Stojanovic, N., Studer, R.: A rule-based language for complex event processing and reasoning. Web Reasoning and Rule Systems. pp. 42–57. Springer Berlin Heidelberg (2010).

19. Hirzel, M., Andrade, H., Gedik, B., Jacques-Silva, G., Khandekar, R., Kumar, V., Mendell, M., Nasgaard, H., Schneider, S., Soule, R., Wu, K.-L.: IBM Streams Processing Language: Analyzing Big Data in motion. IBM J. Res. Dev. 57, 1–11 (2013).

20. Zikopoulos, P.C., Eaton, C., DeRoos, D., Deutsch, T., Lapis, G.: Understanding Big Data. McGraw-Hill (2011).

21. Yao, Y., Gehrke, J.: Query Processing in Sensor Networks. CIDR 2003, Proceedings of the First Biennial Conference on Innovative Data Systems Research (2003).

22. Zadorozhny, V., Chrysanthis, P.K., Labrinidis, A.: Algebraic optimization of data delivery patterns in mobile sensor networks. In Proc. of DEXA (2004).

23. Li, H.-G., Chen, S., Tatemura, J., Agrawal, D., Candan, K., Hsiung, W.-P.: Safety guarantee of continuous join queries over punctuated data streams. Proc. VLDB. (2006).

24. Wolfson, O., Sistla, A.P., Xu, B., Zhou, J., Chamberlain, S.: Domino: Databases for moving objects tracking. In Proc. of the SIGMOD Int. Conf. on Management of Data (1999).

25. Avnur, R., Hellerstein, J.M.: Eddies: continuously adaptive query processing. In Proc. of the Int. Conf. on Management of Data (SIGMOD '00). pp. 261–272. ACM Press, New York, USA (2000).

26. Chakravarthy, S., Mishra, D.: Snoop: an expressive event specification language for active databases. Data Knowl. Eng. 14, 1–26 (1994).

27. Chakravarthy, S., Krishnaprasad, V., Anwar, E., Kim, S.K.: Composite events for active databases: Semantics, contexts and detection. In Proc. of the 20th Int. Conf. on Very Large Data Bases. pp. 606–606. , Santiago, Chile (1994).

28. Zheng, Y., Capra, L., Wolfson, O., Yang, H.: Urban Computing: Concepts, Methodologies and Applications. ACM Trans. Intell. Syst. Technol. 5, 1–55 (2014).

29. Mansouri-Samani, M., Sloman, M.: GEM: A generalized event monitoring language for distributed systems. Distrib. Eng. J. (1997).

30. Rosenblum, D.S., Wolf, A.L.: A Design Framework for Internet-Scale Event Observation and Notification. In Proc. of the 6th European Software Engineering Conference. , Zurich, Switzerland (1997).

31. Yuhara, M., Bershad, B.N., Maeda, C., Moss, J.E.B.: Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages. In Proc. of the 1994 Winter USENIX Conference (1994).

32. Bailey, M.L., Gopal, B., Sarkar, P., Pagels, M.A., Peterson, L.L.: Pathfinder: A pattern-based packet classifier. In Proc. of the 1st Symp. on Operating System Design and Implementation (1994).

33. Gatziu, S., Dittrich, K.R.: Detecting composite events in active database systems using Petri nets. In Proc. of the 4th Int. Workshop on Research Issues in Data Engineering: Active Database Systems. , Houston, Texas, USA (1994).

34. Collet, C., Coupaye, T.: Primitive and composite events in NAOS. Actes des 12ièmes Journées Bases de Données Avancées. , Clermont-Ferrand, France (1996).

35. Bidoit, N., Objois, M.: Machine Flux de Données: comparaison de langages de requêtes continues. 23eme Journees Bases de Donnees Avancees (BDA). , Marseille, France (2007).

36. Gehani, N.H., Jagadish, H. V., Shmueli, O.: Event specification in an active object-oriented database. In Proc. of the ACM SIGMOD Int. Conf. on Management of Data (1992).

37. Pietzuch, P.R., Shand, B., Bacon, J.: Composite Event Detection as a Generic Middleware Extension. IEEE Netw. Mag. Spec. Issue Middlew. Technol. Futur. Commun. Networks. (2004).

38. Yoneki, E., Bacon, J.: Unified semantics for event correlation over time and space in hybrid network environments. In Proc. of the OTM Conferences. pp. 366–384 (2005).

39. Agrawal, R., Srikant, R.: Mining Sequential Patterns. In Proc. of the 11th Int. Conf. on Data Engineering. , Taipei, Taiwan (1995).

40. Giordana, A., Terenziani, P., Botta, M.: Recognizing and Discovering Complex Events in Sequences. In Proc. of the 13th Int. Symp. on Foundations of Intelligent Systems. , London, UK (2002).

41. Wu, E., Diao, Y., Rizvi, S.: High-Performance Complex Event Processing over Streams. SIGMOD (2006).

42. Demers, A.J., Gehrke, J., Panda, B., Riedewald, M., Sharma, V., White, W.M.: Cayuga: A General Purpose Event Monitoring System. CIDR. pp. 412–422. www.crdrdb.org (2007).

43. Balazinska, M., Kwon, Y., Kuchta, N., Lee, D.: Moirae: History-Enhanced Monitoring. CIDR (2007).

44. Gehani, N.H., Jagadish, H. V.: Ode as an Active Database: Constraints and Triggers. In Proc. of the 17th Int. Conf. on Very Large Data Bases. , Barcelona, Spain (1991).

45. Gehani, N.H., Jugadish, H. V., Shmueli, O.: Composite event specification in active databases: Model & Implementation. In Proc. of the 18th Int. Conf. on Very Large Data Bases. , Vancouver, Canada (1992).

46. Gatziu, S., Dittrich, K.R.: SAMOS: an active object-oriented database system. IEEE Q. Bull. Data Eng. Spec. Issue Act. Databases. (1993).

47. Adaikkalavan, R.: Snoop event specification: formalization algorithms, and implementation using interval-based semantics. MS Thesis. Univ. Texas Arlington. (2002).

48. Chakravarthy, S.: SENTINEL: an object-oriented DBMS with event-based rules. In Proc. of the ACM SIGMOD Int. Conf. on Management of Data. , New York, USA (1997).

49. Jakobson, G., Weissman, M.D.: Alarm Correlation. IEEE Netw. 52–59 (1993).

50. Liu, G., Mok, A.K., Yang, E.J.: Composite events for network event correlation. In Proc. of the 6th IFIP/IEEE Int. Symp. on Integrated Network Management. pp. 247–260 (1999).

51. Wu, P., Bhatnagar, R., Epshtein, L., Bhandaru, M., Shi., Z.: Alarm Correlation Engine (ACE). In Proc. of the IEEE/IFIP Network Operation and Management Symposium. pp. 733–742 (1998).

52. Nygate, Y.A.: Event correlation using rule and object based techniques. In Proc. of the IFIP/IEEE Int. Symp. on Integrated Network Management. pp. 278–289 (1995).

53. Appleby, K., Goldszmidth, G., Steinder, M.: Yemanja – A Layered Event Correlation Engine for Multi-domain Server Farms. Integr. Netw. Manag. 7, (2001).

54. Yemini, S.A., Kliger, S., Mozes, E., Yemini, Y., Ohsie, D.: High Speed and robust Event Correlation. IEEE Commun. Mag. 34, 82–90 (1996).

55. Roncancio, C.L.: Towards duration-based, constrained and dynamic event types. In Proc. of the 2nd Int. Workshop on Active, Real-Time, and Temporal Database Systems (1998).

Appendix A

The events e_1 and e_2, used in the following definitions, are occurrences of the event types E_1 and E_2 respectively (with $E_1 \neq E_2$) and can be any primitive or composite event type. An event is considered as *durative*, i.e., it has a duration going from the instant when it starts until the instant when it ends [55] and its occurrence time is represented by a time interval `[startI-e, endI-e]`.

6.1 Binary operators

Binary operators derive a new composite event from two input events (primitive or composite). The following binary operators are defined by most existing event models [34, 45, 46, 55]:

- **Disjunction: $(E_1 \mid E_2)$**
  There are two possible semantics for the disjunction operator "|": *exclusive-or* and *inclusive-or*. Exclusive-or means that a composite event of type $(E_1 \mid E_2)$ is initiated and terminated by the occurrence of e_1 of type E_1 or e_2 of type E_2, whereas inclusive-or considers both events if they are simultaneous, i.e. they occur "at the same time". In centralized systems, no couple of events can occur simultaneously and hence, the disjunction operator always corresponds to exclusive-or. In distributed systems, two events at different sites can occur simultaneously and hence, both exclusive-or and inclusive-or are applicable.
- **Conjunction: (E_1 , E_2)**
  A composite event of type (E_1 , E_2) occurs if both e_1 of type E_1 and e_2 of type E_2 occur, regardless their occurrence order. Event e_1 and e_2 may be produced at the same or at different sites. The event e_1 is the initiator of the composite event and the event e_2 is its terminator, or vice versa. Event e_1 and e_2 can overlap or they can be disjoint.
- **Sequence: $(E_1 ; E_2)$**
  A composite event of type $(E_1 ; E_2)$ occurs when an e_2 of type E_2 occurs after e_1 of type E_1 has occurred. Then, sequence denotes that event e_1 "happens before" event e_2. This implies that the end time of event e_1 is guaranteed to be less than the start time of event e_2. However, the semantics of "happens before" differs, depending on whether composite event is a local or a global event. Therefore, although the syntax is the same for local and for global events, the two cases have to be considered separately.
- **Concurrency: $(E_1 \parallel E_2)$**
  A composite event of type $(E_1 \parallel E_2)$ occurs if both events e_1 of type E_1 and e_2 of type E_2 occur virtually simultaneously, i.e. "at the same time". This implies that this operator applied to two distinct events is only applicable in global events; the events e_1 and e_2 occur at different sites and it is not possible to establish an order between them. The concurrency relation is commutative.

- **During: (E_2 during E_1)**
  The composite event of type $(E_2$ during $E_1)$ occurs if an event e_2 of type E_2 happens during event e_1 of type E_1, i.e. e_2 starts after the beginning of e_1 and ends before the end of e_1.
- **Overlaps: (E_1 overlaps E_2)**
  The beginning of event e_1 of type E_1 is before the beginning of event e_2 of type E_2 and the end of e_1 is during e_2 or vice versa.
- **Meets: (E_1 meets E_2)**
  The beginning of event e_2 of type E_2 is immediately after the end of event e_1 of type E_1.
- **Starts: (E_1 starts E_2)**
  The beginning of event e_1 of type E_1 and e_2 of type E_2 are simultaneous. The occurrence interval of $(e_1$ starts $e_2)$ is [startT-e_1, *latest*(endT-e_1, endT-e_2)].
- **Ends: (E_1 ends E_2)**
  The end of event e_1 of type E_1 and event e_2 of type E_2 are simultaneous. The ends relation is commutative. The occurrence interval of $(e_1$ ends $e_2)$ is [*earliest* (startT-e_1, startT-e_2), endT-e_2].

6.2 Selection operators

Selection operators allow searching occurrences of an event type in the event history. The selection $E^{[i]}$ defines the occurrence of the i^{th} element of a sequence of events of type E, $i \in N$; where N is a natural number greater than 0, during a predefined time interval I. The following selection operators are distinguished in event models such as SAMOS [33, 46]:

- **First occurrence: (*E in I)**
  The event is produced after the first occurrence of an event of type E during the time interval I. The event will not be produced by all the other event occurrences of E during the interval.
- **History: (Times(n, E) in I)**
  An event is produced when an event of type E has occurred with the specified frequency *n* during the time interval I.
- **Negation: (Not E in I)**
  The event is produced if any occurrence of the event type E is not produced (i.e. the event did not occur) during the time interval I.

6.3 Temporal operator

A composite event can be represented by the occurrence of an event and an offset $(E + \Delta)$, for example, $E = E_1 + 00:15$ to indicate fifteen minutes before the occurrence of an event of type E_1. Thus, the occurrence time of E is $[endT\text{-}e_1, endT\text{-}e_1 + \Delta]$.