Extending Complex Event Processing to Graph-structured Information

Gala Barquero
Universidad de Málaga, Spain
gala@lcc.uma.es

Loli Burgueño
Universidad de Málaga, Spain
UOC, Barcelona, Spain
CEA-List, Paris, France
loli@lcc.uma.es

Javier Troya
Universidad de Sevilla, Spain
jtroya@us.es

Antonio Vallecillo
Universidad de Málaga, Spain
av@lcc.uma.es

ABSTRACT

Complex Event Processing (CEP) is a powerful technology in real-time distributed environments for analyzing fast and distributed streams of data, and deriving conclusions from them. CEP permits defining complex events based on the events produced by the incoming sources in order to identify complex meaningful circumstances and to respond to them as quickly as possible. However, in many situations the information that needs to be analyzed is not structured as a mere sequence of events, but as graphs of interconnected data that evolve over time. This paper proposes an extension of CEP systems that permits dealing with graph-structured information. Two case studies are used to validate the proposal and to compare its performance with traditional CEP systems. We discuss the benefits and limitations of the CEP extensions presented.

CCS CONCEPTS

• **Software and its engineering** → *Model-driven software engineering*; *Software performance*; • **Computer systems organization** → *Real-time systems*; • **Information systems** → *Query languages*; *Graph-based database models*;

KEYWORDS

Complex Event Processing, Graph-structured information

ACM Reference Format:
Gala Barquero, Loli Burgueño, Javier Troya, and Antonio Vallecillo. 2018. Extending Complex Event Processing to Graph-structured Information. In *ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS '18), October 14–19, 2018, Copenhagen, Denmark*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3239372.3239402

1 INTRODUCTION

Stream processing systems are becoming widespread due to the steadily growing number of information sources that continuously produce and offer data. Among them, Complex Event Processing (CEP) is currently a mature technology for analyzing and correlating streams of information about real-time events that happen in a system, and deriving conclusions from them [6, 14, 23, 24]. A distinguishing feature of CEP, not present in many stream processing systems, is that it permits defining complex events or patterns on top of the primitive events in order to identify elaborate meaningful circumstances and to respond to them as quickly as possible. Such event types and event patterns are defined using Event Processing Languages (EPLs).

In many applications, however, the information that needs to be analyzed is not structured as a mere sequence of partially ordered timed events, but as graphs of highly interconnected datasets that evolve over time—e.g., contagious disease spreading data or social media networks. In these systems, not only the type of event and the moment in time at which it occurs are relevant, but also its connections with other surrounding objects, the state of these objects, and the current network topology.

This paper proposes an extension of CEP systems that permits dealing with graph-structured information, in addition to their timed-events stream processing capabilities, and its implementation using graph technologies based on cluster computing platforms. Two case studies are used to illustrate the proposal, and to discuss the benefits and limitations of this kind of extension of CEP systems.

Our solution makes use of the fact that the structure of a CEP event stream can be generalized from a sequence of time-ordered elements to a model (i.e., a graph of interrelated elements). Besides, the behavior of a CEP system can be naturally considered as a special case of in-place model transformation (MT) [11, 12], and therefore most of the MT machinery can be reused. To tackle the stringent requirements of CEP systems regarding their scalability and performance, we propose the use of the recent graph-parallel computation technologies (such as the GraphX component of Spark [18]), which provide the required supporting storage and access infrastructure.

One of our major contributions is the generalization of the concept of CEP *windows* to the more general concept of graph *vicinity.* CEP makes use of windows in order to restrict the matching space to the substream of events with a certain *time* or *length* window, i.e., with timestamps inside a temporal window, or limited to a consecutive number of events, respectively. Our approach also permits restricting the pattern matching to a subset of the whole space. In addition to the temporal or length windows from the original CEP, we introduce the concept of *spatial windows*, which are defined in terms of *vicinity graphs* [13], composed of elements related to those in the rules by means of the graph arcs, and which constitute their local context. It is important to note that, in our proposal, time becomes just one of the possible dimensions through which we can navigate the graph.

The structure of the paper is as follows. First, Section 2 briefly introduces CEP systems and their main features, as well as Apache Spark and graph-parallel computation technologies, the three underlying technologies over which our approach is built. Then, Section 3 describes our extension to CEP systems for dealing with graph structures, beyond sequences of timed events. Section 4 discusses how we have implemented our proposal using cluster computing platforms, and Section 5 describes the validation experiments we have performed to assess our approach. Finally, Section 6 relates our work to other similar approaches and Section 7 concludes and outlines some future lines of work.

2 BACKGROUND

2.1 CEP in a nutshell

CEP [14, 23] is a form of Information Processing [6] whose goal is the definition and detection of situations of interest, from the analysis of low-level event notifications [7]. According to the Event Processing Technical Society [15], we use the *simple events* term to refer to the low-level primitive event occurrences, and *complex* events to those that summarize, represent, or denote a set of other events. *Derived* events are a particular kind of complex events, which are generated as a consequence of applying a process or method to one or more other events. In summary, CEP systems analyze streams of *simple* events to detect occurrences of *complex* events, that represent the high-level situations of interest to the CEP modeler, using declarative *rules* that define the derived events in terms of *patterns* of (simple or complex) events, their contents, and temporal relations.

Although several CEP systems and languages exist, they all share the same basic concepts, mechanisms and structure. These are briefly introduced below.

- **Events**. In CEP, every event (simple or complex) has a *type* and a set of *attributes.* Events are atomic, happen instantaneously, and they all have an attribute with information about the moment in time at which they occur.
- **Patterns**. A CEP rule defines a derived event, by means of a *pattern* that combines other events. Whenever the pattern is detected in the stream (i.e., it is *satisfied* by the stream events), the derived event is created. The simplest kind of pattern (called *selection pattern*) permits creating derived events every time a given simple event is detected in the stream.
- **Elements of patterns**. There are several representative elements of CEP patterns that are commonly used in rules.
  - **Windows**: We can assign *windows* to patterns, restricting their scope. Windows could refer to specific time intervals (time windows) or to number of occurrences of particular events (length windows). Moreover, we can distinguish two types of windows: batch and sliding. The first ones have fixed starting and ending points. The latter ones move the interval so its ending point coincides with the current position of the pointer traversing the stream (e.g., the current time).
  - **Temporal sequencing of events**: A key CEP operator is *followedBy* ("`->`"), which introduces a temporal ordering between two events. Events related by this operator do not need to be consecutive: "`A -> B`" only implies that event `A` occurs some time before `B`, i.e., the timestamp of `A` precedes that of `B`.
  - **Pattern combination**: Patterns can be combined in different ways by using logical operators (or, and, etc.) and temporal connectors (until, while, etc.), among others. Negation is also possible, representing the fact that an event has not happened. In addition, windows can be combined, restricting their scope.

Such elements are indeed very close to those of in-place model transformation rules, with two main additions. First, given that the stream can be considered infinite, or at least too large to be handled in full, CEP patterns use the concept of *window* to restrict their scope. Therefore all matches are *local*, within the scope defined by the window. Second, all events have a timestamp and therefore they can be (totally) ordered; in other words, the stream can be considered as a linear sequence of ordered events. Relationships between the events are not explicitly modeled, they are normally specified either by time-order information, or by name/id matching.

2.2 Apache Spark

Apache Spark is a general-purpose cluster computing platform that extends the popular MapReduce model to efficiently support more types of computations, including interactive queries and stream processing [20]. One of the main features Spark offers for speed is the ability to run computations in memory, while being more efficient than, e.g., MapReduce for complex applications running on disk.

Spark combines different processing types and it is designed to be fast, highly accessible, and provide simple APIs in different programming languages such as Python, Java, Scala, and SQL, and a set of rich built-in libraries. It also integrates smoothly with other Big Data tools.

Spark is built on the concept of distributed datasets, which contain arbitrary Java, Python or Scala objects, upon which parallel operations can be efficiently performed. The Spark API is built around the concept of a *resilient distributed dataset* (RDD), which is a fault-tolerant collection of elements that can be operated on in parallel. The RDD API supports two types of operations: transformations, which define a new dataset based on previous ones, and actions, which execute operations on a cluster. RDDs can be created in two ways: parallelizing an existing collection, or referencing a dataset in an external storage system (e.g. as a shared filesystem, HDFS, HBase, or any other data source offering a Hadoop Input-Format).

Spark also provides different tools that are built as libraries on top of Spark, such as *Spark Streaming*, which enables processing of live streams of data; *Spark SQL*, for working with structured data; *MLib*, containing common machine learning functionality, and *GraphX* for manipulating graphs.

2.3 Graph processing systems

Graph structures are commonly used for representing the data of models, and for performing patterns on them. Graphs are composed of nodes, edges and properties for this representation. Nodes represent objects in a domain of interest, and edges represent relationships between these objects. Both nodes and edges can be typed, and may also have attributes, which are also typed. Relationships can be directional or bidirectional, and they can also be derived.

Graph patterns are structures that should be matched against the graph for performing queries. Graph patterns can use variables, and be augmented with other (relational-like) features, such as projections, unions, optionals, and differences.

Graph databases [29] can be very efficient for performing path queries, and hence they have been effectively used for storing and performing operations on very large models [9]. However, the stringent requirements imposed by real-time event processing systems requires in-memory storage of data and very fast processing frameworks. In fact, as discussed later in Section 5, our first prototypical implementation of our approach was developed using Neo4j and Cypher. However, the performance penalty introduced by the use of disk storage resulted in unacceptable response times when compared with native CEP systems—which are at least one order of magnitude faster.

This is where Graph Processing Systems, such as Pregel [25] or PowerGraph [17] can be very efficient. In a nutshell, graph processing systems define computation at the granularity of vertices and their neighbors, and exploit the sparse topology of the graph. These systems can naturally express and efficiently execute graph algorithms such as PageRank [26] or community detection [22], on graphs with billions of elements [18].

In contrast, general-purpose distributed dataflow frameworks (such as Map-Reduce, Spark or Dryad) provide efficient dataflow operators (e.g., map, reduce, group-by, join), and define computation as dataflow operators at either the granularity of individual items (e.g., filter, map) or across entire collections (i.e., operations like non-broadcast join that require a shuffle). These frameworks are very well suited for analyzing unstructured and tabular data.

A recent trend for efficient graph processing combines both technologies. One exponent of this trend is GraphX [18], an efficient graph processing framework embedded within the Spark distributed dataflow system. GraphX is built as a library on top of Spark. It encodes graphs as horizontally-partitioned collections and then expresses the GraphX API for graph computation in terms of standard dataflow operators on these collections (e.g., join, map, group-by). A series of internal optimizations are used to improve the performance of the API operations, making it comparable to (or even faster than) other specialized graph processing platforms.

3 EXTENDING CEP

3.1 A Running example

To motivate and illustrate our proposal, consider a system that needs to take into account the information provided by Flickr[1] and Twitter[2], and use it together to identify some situations of interest. The metamodel for these two information systems is depicted in Figure 1. Note the existence of only one common class, `Hashtag`. The rest of the elements could be related, but there is no automated manner to implement such a relation that is 100% reliable (for example, automatically relating users based on their Twitter and Flickr ids is impossible).

Given such a system, we are interested in identifying some situations of interest to its users, and react to them. Examples of these situations are the following:

S1: A hashtag has been used by both Twitter and Flickr users at least 100 times in the last hour. We would like to generate a new `HotTopic` object in the graph that refers to this hashtag.

S2: The hashtag of a photo is mentioned in a tweet that receives more than 30 likes in the last hour. A `PopularTwitterPhoto` element is created.

S3: A photo is favored by more than 50 Flickr users who have more than 50 followers. A `PopularFlickrPhoto` element is created.

S4: A user, with an *h-index*[3] higher than 50, posts three tweets in a row in the last hour containing a hashtag that describes a photo. In this case, we generate a `NiceTwitterPhoto` object.

S5: A Flicker user favorites a photo identified as `NiceTwitterPhoto` but none of her followers do. If so, we generate a `Misunderstood` object identifying the user.

S6: Taking into account the 10, 000 most recent tweets, a user with h-index higher than 70 and more than 50K followers who writes a tweet raises an `InfluencerTweeted` event.

To record such new objects (that correspond to CEP derived events), we have included in the metamodel 6 extra classes (shaded in gray). Therefore, a model of this application will contain both the information from the sources (Twitter and Flickr) and also the information we generate during its lifetime, as the system evolves.

Note that in order to respond to these patterns, we need to take into consideration not only the more static information about the social network users, but also the dynamic and rapidly changing information appearing such as tweets or likes. Some situations (S4 and S5) also imply traversing the model through its relationships, including the closure of a relationship (e.g., `Follow`). This means that the search space can be extremely high. Other situations (S5) require negations, which also imply searches in a very large model.

3.2 Extending CEP elements and operators

3.2.1 Extending the structure. In order to extend the structure of CEP, we make use of conceptual modeling concepts for representing information items and their relationships, i.e., our information system will be represented by a model, and not just by a sequence of events. Elements of such a model that represent CEP events will

[1]https://www.flickr.com/
[2]https://twitter.com/
[3]In Twitter, a user has an *h-index* of h is she has at least h followers who have more than h followers each.

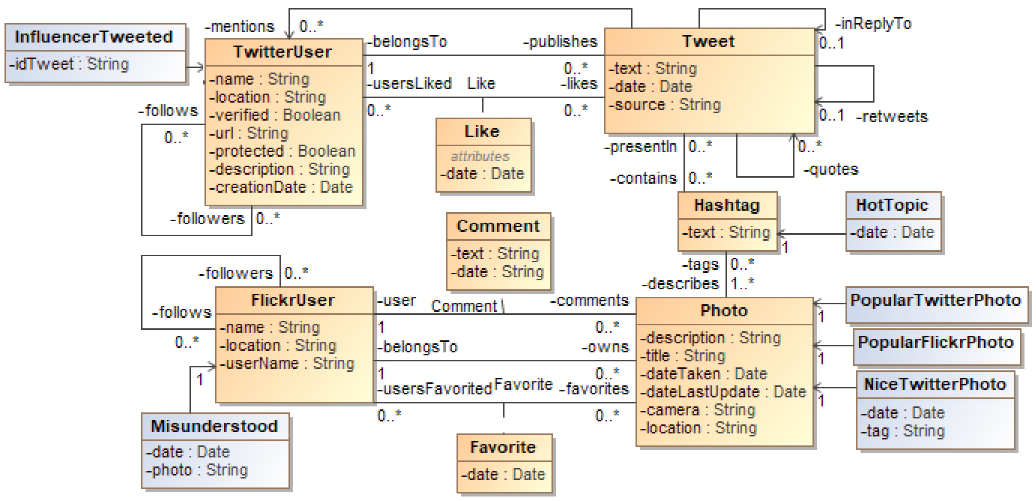

Figure 1: Twitter and Flickr joint Metamodel.

have an attribute, *timestamp*, with information about the moment in time at which the event occurs.

Apart from the relationships defined among the nodes, a derived directed relation `FollowedBy` can be defined between the model elements (i.e., graph nodes) with timestamps if the timestamp of the first element is less than that of the second. In this way, a CEP event stream is just a graph with one single dimension. In addition, the lifecycle of events should be considered too, by means of mechanisms that permit inserting the objects that represent the events when they occur, and removing them when the become outdated (not part of any CEP window).

3.2.2 Vicinity: extending CEP Windows. In CEP, windows could refer to specific time intervals or to number of occurrences of particular events. There are two types of CEP windows: batch and sliding. The first ones have fixed starting and ending points. The latter ones move the interval so its ending point coincides with the current position of the pointer traversing the stream (e.g. the current time). In other words, CEP windows define *segments* of the linear stream of events that determine the *local* context of a match.

When the linear nature of the event stream is generalized to a graph, instead of stream segments (time or length windows) we can use *vicinity graphs* [13], which are subgraphs composed of elements related to one or more objects. They represent the *neighborhood* of these objects, and can be expressed by means of graph patterns.

In this way, we can talk about *spatial* vicinity, in addition to the *temporal* vicinity used in CEP. Then, given one object, its vicinity will be composed of other objects related to it by links (directly, or through a sequence of consecutive links, that we shall call *hops*), i.e., its nearby objects. To determine the window that establishes the local context of a rule, we can simply define the vicinity of the objects in the rule.

There are different strategies to define the vicinity graph of an object, depending on how we navigate through the graph structure, and the goal we pursue. Representative examples of algorithms for creating relevant vicinity graphs of nearby objects are used for finding related pages in the WWW [13, 21]. These algorithms use different strategies, e.g., going through the parents and children of a page, and then visiting the children and parents of those—using a backward-forward and forward-backward strategy. We could also traverse the graph moving only forward or backward (as in the case of CEP sliding windows), or using any other traversal strategy: in-breadth, in-depth, topological, hybrid, etc. Travesal could be done through any kind of link, or we could navigate the graph through some selected kinds of relations. The information on the nodes could also be considered when building the vicinity graph, assigning weights to nodes and to links, instead of just bean-counting them.

Note that if we collapse a graph into the one-dimensional sequence that represents a CEP event stream, by focusing only on the timestamp attribute of the objects that represent events, and the `followedBy` derived relation, the concept of vicinity faithfully corresponds to the CEP concept of window, time becoming just one of the possible dimensions by which we can navigate the graph.

3.2.3 Extending the behavior. Given that now we have a model (i.e., a graph), recursive in-place model transformation rules can be naturally used to represent how derived information items (e.g. complex events) are created and added to the system, hence being able to extend the traditional CEP behavior. Thus, each CEP rule is mapped to an in-place model transformation rule that specifies the matching pattern and creates the derived event.

Non-timed derived events can also be generated by other MT rules, permitting further situations of interest to be detected as the graph evolves—i.e., new nodes or links are created or destroyed.

Finally, to simulate the lifecycle of CEP events, another model transformation rules can be in charge or removing those model elements that represent CEP events with 'old' timestamps—i.e., once they are outside the temporal windows.

4 IMPLEMENTATION

4.1 Architecture

In order to implement and validate our proposal, in the first place, we have used Apache Spark to process the models. To create the graph dataset, a Spark RDD is used for the vertices, and another one for the edges, thus storing the simple events and the relations among them in two RDDs. The graph dataset is populated using the corresponding APIs from the different sources; in our example, the Twitter and Flickr APIs. A thread is in charge of obtaining the batches of raw data, giving them the proper format of RDD, and storing them in the dataset.

CEP patterns are implemented in terms of Spark and GraphX functions written in Scala—which perform the matching process of the rule— followed by the generation of the new objects—, which correspond to the derived elements. We use one dedicated thread per rule that is continuously running, and each rule has a corresponding RDD to store the derived events.

To represent the objects in the graph dataset, we need to distinguish between *transient* and *permanent* information. The former refers to the time-sensitive information coming from sources with high-volume of data that do not need to persist in memory for a long time. Basically, this kind of information corresponds to traditional CEP data streams; once they are analyzed by the CEP rules, they can be discarded. In our example, these can be the tweet feeds. Permanent (or static) information corresponds to data which is stable in time, and that we want to persist in the database for longer periods. For instance, the information about the Twitter and Flickr users, or the Flickr photos, in our example.

Both kinds of data are stored in the dataset, although the manner in which their lifecycles are managed is different. Objects representing permanent information are stored as single nodes in the graph dataset, and relationships among them as relationships in the dataset, too. Transient data (i.e., events) need to be processed as soon as possible by the CEP rules, and each rule should try to be as efficient as possible. Then, we simulate a time window and every object that represents a time-sensitive event has a timestamp and it will be removed when its timestamp expires. The time window value will be set by the rule with the biggest time window, 60 minutes in our example.

When appropriate, our implementation can consider complex events like transient data too, so they will have a timestamp attribute and will be removed when they expire. However, if the same complex event is detected in a time window several times, for efficiency reasons its timestamp will be updated instead of duplicating the event, or discarding the last detection.

4.2 Writing the patterns for the example

As mentioned above, CEP patterns can be implemented by means of in-place model transformation rules, which in turn we express in this version as Spark patterns.

To illustrate how these patterns are expressed, in this section we show some of the patterns that correspond to the situations described in the running example. The rest of them, together with the rest of the project’s models, artifacts and programs, can be accessed and downloaded from our Git repository [3] and the project website [2].

In the first place, pattern **S1** generates a new `HotTopic` object in the dataset every time a hashtag is used by both Twitter and Flickr users at least 100 times in the last 60 minutes. This can be expressed in terms of a Spark pattern as the Listing in Fig. 1 shows.

In the first part, the rule selects groups of three objects (`Hashtag`, `Tweet`, and `Photo`) that are related by the incoming `Hashtag` associations (`contains` and `tags`). Then, our code computes the incoming links for each node, and selects those results that have more than 100 incoming links. Finally, it creates a `HotTopic` object for each matching, and stores it in a Spark RDD with a timestamp and the `id` of the `hashtag` attribute.

Note how in the last instruction, the pattern checks if the new detected event is in the RDD and in that case updates the dataset with the new event. In turn, the last filter discards all `HotTopic` events created more than one hour ago (3600000 milliseconds), as mentioned in the description of the event (Sect. 4.1).

Pattern **S4** (whose code is not listed here) requires a user, with an *h-index* higher than 50, to post three tweets in a row containing a hashtag that corresponds to a photo in the last hour. If such a situation is detected, a `NiceTwitterPhoto` object identifying the photo is created. Observe how this pattern specifies that the user *publishes* a tweet that *contains* a hashtag. In this case we use a 2-*hops spatial* window, meaning that we will traverse all objects of class `TwitterUser` that have a direct relationship to class `Tweet`, and the `Tweet` class is directly connected with class `Hashtag`, but there is not a direct relation between `TwitterUser` and `Hashtag` classes. Additionally, the pattern checks that the hashtag *describes* a photo, therefore it uses 3 hops since it will traverse relation *describes* or *tags* in the metamodel.

In pattern **S5**, a Flickr user favorites a photo which has been identified as `NiceTwitterPhoto`, but none of her followers does. In this case, we generate a `Misunderstood` object identifying user. This pattern shows an example of a NAC (Negative Application Condition) and complex event from another complex event.

Finally, pattern **S6** creates `InfluencerTweeted` events considering the 10K most recent tweets. For this we need a length window of tweets and an aggregation function for calculating the users’ *h-index*.

As a final remark, note that in this paper we are just concerned with the expression of such patterns, not on how to determine the optimal size of the spatial or temporal windows. These are decisions that depend on the problem domain and on the particular patterns, and therefore they fall outside the scope of this paper.

5 DISCUSSION

Once we are able to write CEP patterns in terms of Spark ones, this section discusses some aspects of our proposal, as well as some of its main benefits and limitations.

5.1 Performance

5.1.1 Comparison with CEP solutions. Performance is a key issue in data processing systems, specially when dealing with high volumes of information. CEP systems are optimized for this, and permit processing large streams of events very efficiently. Apart from the internal optimizations to handle the events in parallel, they normally use in-memory structures for storing the events

Listing 1: Scala code for the HotTopic event.

```
def hotTopic {
  // Filters photos, tweet, hashtags and incoming links to hashtags
  val hashtagTweetPhotoFilter = graph.subgraph(
    vpred = (id, attr) =>
      (  attr.isInstanceOf[Hashtag]
      || (attr.isInstanceOf[Tweet] && (System.currentTimeMillis() - attr.asInstanceOf[Tweet].date) < 3600000)
      || (attr.isInstanceOf[Photo] && (System.currentTimeMillis() - attr.asInstanceOf[Photo].dateLastUpdate) < 3600000)
      ),
    epred = t =>
      t.toString().contains("contains") || t.toString().contains("tags")
  )
  // Computes incoming links to each node
  val hashtagTweetPhotoFilterInDegrees =
    hashtagTweetPhotoFilter.outerJoinVertices(hashtagTweetPhotoFilter.inDegrees) {
      (id, _, counter) => counter match {
          case Some(counter) => counter
          case None          => 0 // No inDegree means zero inDegree
       }
    }
  // Gets Hashtags with more than 100 incoming links. Result: HotTopic graph with Hashtags
  val graphHotTopicAux = hashtagTweetPhotoFilterInDegrees.subgraph(
    vpred = (id, attr) => attr >= 100).mapVertices((id, attr) => (System.currentTimeMillis())
  )
  //Result
  verticesHotTopic = (graphHotTopicAux.vertices.map(rdd =>
    (rdd._1, new HotTopic(rdd._1, rdd._2))) ++ verticesHotTopic).reduceByKey((a, b) =>
      if (a.currentTimestamp > b.currentTimestamp) {a} else {b})
      .filter { v => (System.currentTimeMillis() - v._2.asInstanceOf[VertexProperty].currentTimeStamp) < 3600000
      }
  }
```

being processed. This is possible because every CEP pattern only deals with a substream of events (those inside the pattern window) whose size is bounded.

However, in our case we need to deal with much larger models, because the graph should contain not only the portion of the *transient* data, but also the application's *permanent* information—note that we cannot deal with the events independently, because one of the assumptions in our case is that they are highly connected to other objects. Hence, the performance of the kinds of applications that we have in mind can not be expected to be as optimal as conventional CEP systems. However, we are interested in knowing the performance penalty that we obtain if we had to deal with similar data, i.e., our overhead. In other words, we are interested in the following research question:

> **R1**. Is the performance of our approach similar to the performance of existing off-the-shelf CEP engines, when dealing with just streams of information (i.e., with no graph structure)?

To respond to this question we have conducted one experiment, consisting of replicating one typical CEP application using our approach, and measuring the overhead that we obtain in the response times with regards to those obtained by the CEP system.

We conducted two versions of this experiment. The first one consisted in sending a sequence of events to our system and to the CEP system, every time they occurred. In this case, we were simulating a realistic behavior of the applications. Secondly, we sent the same sequence, but all at once, i.e., without any delay between the events. All tests were run on a machine whose operating system is Ubuntu 64 bits, with 64Gb of RAM memory, and an Intel Xeon CPU E5-2680 processor with 16 cores of 2.7 GHz. We used Spark version 2.3.0 in our implementation, and Esper version 4.7.0[4] as the CEP engine.

In CEP, performance is mainly dependent on the window size defined for the patterns. Therefore we checked with different window sizes (note that, as mentioned in Section 4.1, only events with timestamps inside the windows are maintained in the dataset).

The example used to conduct the tests[5] assumes a fleet of motorbikes equipped with sensors that produce real-time information about their state, including the timestamp of the event, the motorbike id, the name of the current location, the speed in Km/h, the pressure of the two tires measured in BARs, and whether the driver is on the seat or not. Listing 3 shows some tuples which are examples of these simple `Motorbike` events. Timestamps are expressed using the POSIX time convention, which is roughly the number of seconds that have elapsed since January 1, 1970 [19].

Listing 3: Simple events

```
      Motorbike(1520674009,1,"Seville",100,3.1,3.1,true)
      Motorbike(1520674010,1,"Seville",90,3.1,3.1,true)
      Motorbike(1520674008,2,"Malaga",62,3.01,3.01,true)
      Motorbike(1520674011,1,"Seville",0,3.1,3.1,true)
      Motorbike(1520674009,2,"Malaga",70,3.01,3.01,true)
      Motorbike(1520674012,1,"Seville",0,3.07,3.07,false)
```

We are interested in monitoring these periodic events, generating derived events when some situations of interest occur. In particular, we are interested in the following CEP patterns:

- **BlowOutTyre**: The pressure of one of the tyres of a moving motorbike goes down from more than 2.0 BAR to less than 1.2 BAR in less than 5 seconds.

[4]http://www.espertech.com/esper/
[5]The complete description of the example is available from http://atenea.lcc.uma.es/index.php/Main_Page/Resources/CEP

.

Listing 2: GraphX/Scala code for the DriverLeftSeat pattern.

```
def driverLeftSeat(){
  // Group events by motorbikeId
    val groupMotorbike = graph.vertices.groupBy(attr => attr._2.asInstanceOf[MotorbikeEvent].motorbikeId)
    // Sort events by timestamp
    val groupMotorbikeOrder = groupMotorbike.map(f =>
      (f._1, f._2.toList.sortBy(f => f._2.asInstanceOf[MotorbikeEvent].currentTimestamp))).collect
    // Add new events to dataset DriverLeftSeat
    verticesDriverLeftSeat = verticesDriverLeftSeat ++ groupMotorbikeOrder.flatMap(f => isDriverLeftSeat(f._2))
    val driverLeftSeatVertices = sc.parallelize(verticesDriverLeftSeat).zipWithIndex().map(r =>  (r._2.asInstanceOf[VertexId], r._1))
    //Filter events that are in the defined window
    verticesDriverLeftSeat = verticesDriverLeftSeat.filter{ attr => (System.currentTimeMillis() - attr.currentTimeStamp) < 3000 }
    graphDriverLeftSeat = Graph(driverLeftSeatVertices, sc.parallelize(ListBuffer[Edge[(Long)]]()))
}
//Function for checking complex event
def isDriverLeftSeat(list: List[(VertexId, VertexProperty)]) : ListBuffer[(VertexProperty)] = {
  val logger: Logger = LoggerFactory.getLogger(this.getClass)
  var result = ListBuffer[(VertexProperty)]()
  for (a <- list; b <- list) {
    if (a._2.asInstanceOf[MotorbikeEvent].seat == true && b._2.asInstanceOf[MotorbikeEvent].seat == false
       && b._2.asInstanceOf[MotorbikeEvent].currentTimestamp > a._2.asInstanceOf[MotorbikeEvent].currentTimestamp
       && sc.parallelize(verticesDriverLeftSeat).filter(attr =>
         attr.asInstanceOf[DriverLeftSeatEvent].currentTimestamp1 == a._2.asInstanceOf[MotorbikeEvent].currentTimestamp
         && attr.asInstanceOf[DriverLeftSeatEvent].currentTimestamp2 == b._2.asInstanceOf[MotorbikeEvent].currentTimestamp).count == 0) {
            val driverLeftSeatEvent = new DriverLeftSeatEvent(
                   System.currentTimeMillis(),
                   a._2.asInstanceOf[MotorbikeEvent].motorbikeId,
                   b._2.asInstanceOf[MotorbikeEvent].location,
                   a._2.asInstanceOf[MotorbikeEvent].seat,
                   b._2.asInstanceOf[MotorbikeEvent].seat,
                   a._2.asInstanceOf[MotorbikeEvent].currentTimestamp,
                   b._2.asInstanceOf[MotorbikeEvent].currentTimestamp)
             result += driverLeftSeatEvent
    }
  }
  return result
}
```

- **Crash**: The speed of a motorbike goes from more than 50 km/h to 0 km/h in less than 3 seconds.
- **DriverLeftSeat**: The seat sensor detects that the driver has left the seat.
- **OccupantThrownAccident**: A motorbike suffers a blow out of one of its tyres, then a `Crash` event is detected, and the driver is thrown out, everything within less than 3 seconds.

To show how these patterns are mapped into Spark patterns, Fig. 2 shows the Scala code corresponding to pattern `DriverLeftSeat`, whose expression in Esper is shown in Fig. 4.

.

Listing 4: Esper code for the DriverLeftSeat pattern.

```
@Name('DriverLeftSeat')
insert into DriverLeftSeat
select current_timestamp() as timestamp,
       a2.motorbikeId as motorbikeId,
       a2.location as location
from pattern [every
  a1=Motorbike(a1.seat) ->
  a2=Motorbike((not a2.seat) and (a1.motorbikeId=a2.motorbikeId))]
```

The performance figures obtained for experiments are shown in Tables 1 and 2.

Table 1 compares the results of the Esper system and our implementation for a different number of input elements, using a realistic simulation between motorbike events whereby each motorbike generates one event every second. Both systems are able to process the events with almost no delay.

Table 1: Performance Figures for the Motorbike Example with stream simulation (in seconds).

#Events	5K	10K	20K	30K
Esper	5,001	10,001	20,001	30,002
Spark	5,001	10,002	20,001	30,001

Table 2: Performance Figures for the Motorbike Example without stream simulation (in milliseconds).

#Events	5K	10K	20K	30K
Esper	1,214	2,208	3,610	5,386
Spark	31,039	60,472	119,038	176,424

In turn, Table 2 compares results of Esper system and our implementation for a different number of input elements, when we feed the complete stream of events to the system without any delay between them. We can see that our system takes approximately 6 milliseconds to process each event, whereas Esper only takes some nanoseconds per event.

In any case, as mentioned above, we do not try to compete with Esper nor other dedicated CEP systems, because in order to be able to deal with graph-structured information and high volumes of data, we have to pay the price of storing a large amount of data. Even so, for some kinds of applications where the event occurrence

Table 3: Performance Figures for the Twitter and Flickr example (in milliseconds).

	127K/6500K	260K/14000K	554K/28000K
HotTopic	62.00	45.00	35.66
PopularTwitterPhoto	69.00	77.00	68.00
PopularFlickrPhoto	76.67	51.33	55.67
NiceTwitterPhoto	108.00	101.66	107.67
Misunderstood	46.33	57.33	53.66
InfluencerTweeted	226.00	334.33	424.33

rate is not very high (e.g., more than one event per millisecond), the response time of our solution is acceptable.

5.1.2 Absolute performance. Apart from estimating the potential overhead of our solution when compared to pure CEP systems, we were also concerned about the response times of our prototype implementation.

> **R2**. Is the performance of our approach acceptable for dealing with large datasets?

Table 3 shows the response times we have obtained when running the Flickr and Twitter example with different patterns and datasets of different sizes. Columns headers indicate the number of nodes/associations of the models used in each test. For example, the first one consisted on 127,000 objects and 65 millions of associations (links) among them, which accounts for a total of 6.627 millions of elements in the graph. Columns 2 and 3 correspond to models with 14.26 and 28.554 million elements, respectively. The high number of links between the model objects reflects the high number of possible relations defined in the metamodel (Fig. 1). Executions were carried out using the same hardware as we did for the motorbike experiments. To conduct this experiment we developed a program that populates our dataset with a specific number of events (coming from synthetic data previously created) and then run the patterns in parallel. Roughly, the ratio of transient to persistent data in these models is 1:11. The complete code for these patterns, and the rest of the artefacts, can be found at [2].

In general, most results are in the range of milliseconds, something that can be acceptable for applications with a non very high event occurrence rate, as explained before.

Furthermore, in a previous approach we used Neo4j and Cypher, respectively, to represent the graph dataset and to express the queries. Although the usability of that approach was much better than the one we obtain with GraphX (see next section), the performance was not acceptable: on the one hand, the response times were of two orders of magnitude higher with respect to the ones we get with GraphX (minutes instead of milliseconds); and on the other hand, the number of nodes we were able to handle in reasonable time was below 20K (whilst now we are able to deal with models of 28 millions elements in a reasonable time). This is why we decided to abandon the Neo4j path and to use Spark.

5.2 Expressiveness

In addition to getting an initial assessment of the performance of our proposal, we also wanted to check its expressiveness for describing the kinds of queries that we initially devised.

.

Listing 5: Cypher code for the DriverLeftSeat pattern.

```
match
  (m1{seat:'true'}), (m2{seat:'false'})
where
  m1.id=m2.id and (m2.timestamp−m1.timestamp)=1
create
  (m1)<−[:EVENT]−(:DriverLeftSeat{ timestamp:m2.timestamp,
          id:m2.id,location:m2.location})−[:EVENT]−>(m2)
```

> **R3**. How expressive is our proposal with regard to CEP, i.e., can we write all kind of CEP patterns with GraphX?

We still need to perform a detailed analysis to see if all CEP patterns can be expressed with our proposal, but initially GraphX is based on a general purpose and very expressive language like Scala. For now, we have been able to use Scala to express the patterns for the case studies used to validate our proposal. However, we plan to study the benefits that developing a domain-specific language (DSL) could bring along. Such a DSL would be closer to the domain expert and tailored at facilitating the definition of graphs and queries that use vicinity windows, thus improving the expressiveness.

> **R4**. How easy is it to use our proposal, i.e., how easy is it to write our patterns with GraphX?

This is what we consider one of the weakest aspects of our proposal at this moment. Writing GraphX patterns in Scala is not a trivial task. For instance, compare the code shown in figures 2 and 4, corresponding to the `DriverLeftSeat` pattern expressed in GraphX/Scala and in Esper, respectively. It is easy to see the benefits of using a dedicated language such as Esper, specifically tailored to express these kinds of patterns.

Something that we also realized when writing the patterns is that most model transformation languages may have difficulties specifying spatial windows in a natural manner. This is not the case for temporal windows, since we could use the object's timestamps to restrict the selection and matching processes. However, graph database languages such as Cypher or Gremlin provide very interesting mechanisms for establishing spatial windows, and greatly simplify the expression of the patterns we are interested in. For example, Figure 5 shows the `DriverLeftSeat` pattern expressed in Cypher.

We are working (as part of our future work) on an automatic mapping from Cypher to GraphX, which will allow users to write their patterns in Cypher, and then get the corresponding GraphX queries. Note that Cypher has a very special feature for representing spatial windows in a very easy way, since it is possible to specify both a specific number of hops (over one type, or over *any* type of relation in the graph), or a bound on it, in the query [29]. Extending model transformation languages with these mechanisms could be an interesting line of research, too. Finally and as mentioned before, we plan to study the possibility to develop a DSL to express patterns and queries, and which would be mapped to Graphx concepts.

5.3 Correctness

The use of temporal or spatial windows provides a very interesting mechanism for tackling scalability issues. In CEP systems, temporal windows permit dealing with the infinite nature of event streams, restricting the queries to the events in the window and ignoring the rest. Similarly, restricting model queries to bounded sets of elements and their relations (i.e., the submodels determined by the spatial windows) permit alleviating the severe scalability problems of queries on huge models. The problem here is that we are trading performance for correctness, because the results of the queries on these reduced models may deviate from those obtained when querying the complete model. Of course, the larger the window the more accurate the query results—but also at a higher performance cost. One essential issue here is to be able to estimate the error made by these approximations [28]. Such estimations fall out of the scope of this paper, and will be part of our future work.

5.4 Threats to Validity

According to Wohlin et al. [30], there are four basic types of validity threats that can affect the validity of our study. We cover each of these in the following paragraphs.

Construct validity — how accurate is the relationship between theory and what is observed?

A possible construct validity threat, known as the mono-method bias, is related to the use of one single metric for measuring performance. In particular, we have measured the execution time with the different technologies, queries defined, and input models of different size. Other measures for performance exist such as throughput or instruction per cycle. However, we believe response time is the most intuitive and easily understandable in our context.

Conclusion validity — are there factors that might affect the ability to draw correct conclusions from the data obtained from the experiments?

The execution times obtained in all experiments can be influenced by the transitory load of the machine where they have been obtained. In order to mitigate this threat, we have taken the average execution time of a number of runs. Furthermore, all experiments have been run on the same machine and under the same circumstances and time range.

Internal validity — are there factors which might affect the results in the context of the case study?

Concerning the performance figures, we would need to experiment with more case studies to confirm that the processing times of our proposal are acceptable when dealing with very large datasets of different kinds. Of course, we can never compete with Esper or any other CEP engine because they only deal with *transient* information. In the kinds of applications we are interested in, the *permanent* data needs to be managed, too, and therefore we need to be able to store it and navigate through the model's relationships in the patterns. This imposes some performance penalties, but we have seen how our proposal has much better performance results than other solutions based on graph databases, and the results we obtain are still acceptable.

The performance figures obtained also depend on the concrete expression of the patterns. Therefore, there might be alternative ways of writing the same patterns which would yield better response times. In any case, our current figures are reasonable enough to show the feasibility of our proposal.

With respect to the expressiveness of our proposal, we would need to conduct more studies, although our initial experiments have shown that the use of general-purpose languages such as Scala and GraphX are expressive enough. There is the issue of how easy it is to write queries with our proposal, since the more complex the patterns are, the more difficult they become to debug and prove correct. In order to mitigate this threat, we plan to develop as future work a domain-specific language close to the domain expert where queries can be more easily expressed. This DSL would be then automatically translated into Graphx for execution.

External validity — to what extent is it possible to generalize our findings in general?

So far, we cannot claim any performance results outside the context of the presented case studies. Nevertheless, the evaluation method used in the case study can indeed be applied on other examples as well. In addition, the case study may be repeated on other hardware platforms to analyze, e.g., the impact of the number of cores or the memory size on the performance.

6 RELATED WORK

Our work is mainly related to CEP systems (e.g. [6, 7, 14]), although they do not provide support for dealing with graph-structured information.

Probably the closest work to ours is [12], where the authors use VIATRA to implement a very efficient CEP system using incremental model queries techniques [27] with reactive transformations [11]. The event stream is populated from elementary model changes by the incremental query engine, and then the CEP engine is in charge of identifying complex the events that are used to trigger the execution of the model transformation rules.

We depart from that work in three main aspects. First, they base their approach on the representation and handling of the incremental changes of the model, instead of using a state-based as we do here. That is, we represent observations while they represent changes. Second, we make use of current Spark technologies, instead of the traditional MDE technologies, in order to reach the performance and scalability required when very large models need to be efficiently processed. Finally, they do not consider the idea of spatial windows to restrict the model queries, in order to improve performance.

The use of spatial windows in models has been already suggested for dealing with infinite [4] or streaming models [5], where only a portion of elements (e.g., the ones inside a *sliding window*) is available at any given moment in time, or in the case of *approximate* model transformations [28]. However, these approaches did not satisfactorily resolve one important issue: they had to consider that some connections between model elements could be broken because not all the elements within a window are available at the same time, and therefore the models could not be correct. Here we have used the concept of vicinity graph, which permits tacking these issues, although also with some associated costs: as in CEP systems, the use of spatial windows trade efficiency for accuracy, given that decisions are made based on the local context of the query, and not on the complete model.

Different graph query languages exist in the model-driven community, which permit specifying graph patterns to define graph queries. GROOVE [16], Henshin [1] and other TGG tools, and VIATRA [11] are example of these languages.

Regarding the use of graph databases for storing large models, NeoEMF [9] provides a multi-database model persistence framework for very large models, and [10] defines a language to perform OCL queries on graph databases that outperforms (in terms of memory footprints and time) other existing solutions. In addition, [8] defines a mapping from ATL to Gremlin, which enables model transformations on large models stored in graph databases. As mentioned above, we tried with Neo4j and also with Gremlim, but the results were not good enough when compared to CEP systems.

7 CONCLUSIONS AND FUTURE WORK

This work has explored the extension of CEP concepts and mechanisms applied to real models that contain graph-structured information, beyond sequential streams of events. The concept of *vicinity* has been introduced, and a prototypical solution has been developed to validate the proposal and to analyze its advantages and limitations.

The proposal has just set the basis and initial ideas for this line of work, and opens the path for different research alternatives. In particular, we are interested in investigating what kinds of changes in the graph structure, patterns and in the architecture of our prototypical implementation would serve to optimize the response time of our current solution. Furthermore, a possible next step for improving the response time could be implementing Spark Streaming in our system, and combining it with Graphx and Spark SQL to optimize the patterns.

We are also planning to develop a mapping from Cypher to GraphX, so that patterns could be expressed in Cypher, which is much more compact, easy to use, and specific to this type of queries, and then translated into the more efficient GraphX queries. Another possibility worth taking into consideration is the development of our own domain-specific language (DSL) close to the domain expert knowledge, with the appropriate mechanisms to express queries in a more natural and lightweight way. This DSL would also have the corresponding mapping to Graphx, so that the queries expressed can be executed.

Regarding the concept of vicinity, we are planning to extend MT transformation languages, and in particular ATL, with the concept of local context of a query (i.e., vicinity), and then generate the queries, in a similar way to what [8] do. Moreover, we would like to study how different strategies (in the style of [13, 21]) could be defined to improve the accuracy of the results, and how to measure the error (or lack of accuracy) we are incurring in by selecting a given spatial window when specifying rules that contain NACs, or when restricting the queries to small windows.

Finally, many more experiments and case studies are needed to better assess the expressiveness and performance of our proposal, and to improve its current results. Although our initial results are currently encouraging, there is room for improvement in several aspects, as we have discussed in this paper.

Acknowledgments. This work was partially funded by the European Commission (FEDER), Spanish Research Projects TIN2014-52034-R and TIN2015-70560-R, and the Andalusian Government Project COPAS (P12-TIC-1867).

REFERENCES

[1] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. 2010. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In *Proc. of MODELS'10 (LNCS)*, Vol. 6394. Springer, 121–135.

[2] Gala Barquero, Loli Burgueño, Javier Troya, and Antonio Vallecillo. Accessed: July 2018. *The Graph-CEP Project.* http://atenea.lcc.uma.es/projects/GraphCEP.html.

[3] Gala Barquero, Loli Burgueño, Javier Troya, and Antonio Vallecillo. 2018. Graph-CEP Git repository. https://github.com/atenearesearchgroup/graphCEP.git. Accessed: July 2018.

[4] B. Combemale, X. Thirioux, and B. Baudry. 2012. Formally Defining and Iterating Infinite Models. In *Proc. of MODELS 2012 (LNCS)*, Vol. 7590. Springer, 119–133.

[5] Jesús Sánchez Cuadrado and Juan de Lara. 2013. Streaming Model Transformations: Scenarios, Challenges and Initial Solutions. In *Proc. of ICMT'13 (LNCS)*, Vol. 7909. Springer, 1–16.

[6] Gianpaolo Cugola and Alessandro Margara. 2012. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.* 44, 3 (2012), 15:1–15:62.

[7] G. Cugola, A. Margara, M. Pezzè, and M. Pradella. 2015. Efficient analysis of event processing applications. In *Proc. of DEBS'15.* ACM, 10–21.

[8] Gwendal Daniel, Frédéric Jouault, Gerson Sunyé, and Jordi Cabot. 2017. Gremlin-ATL: a scalable model transformation framework. In *Proc. of ASE'17.* IEEE Computer Society, 462–472.

[9] G. Daniel, G. Sunyé, A. Benelallam, M. Tisi, Y. Vernageau, A. Gómez, and J. Cabot. 2017. NeoEMF: A multi-database model persistence framework for very large models. *Sci. Comput. Program.* 149 (2017), 9–14.

[10] Gwendal Daniel, Gerson Sunyé, and Jordi Cabot. 2016. Mogwaï: A framework to handle complex queries on large models. In *Proc. of RCIS'16.* IEEE, 1–12.

[11] István Dávid, István Ráth, and Dániel Varró. 2014. Streaming Model Transformations By Complex Event Processing. In *Proc. of MODELS'14 (LNCS)*, Vol. 8767. Springer, 68–83.

[12] István Dávid, István Ráth, and Dániel Varró. 2018. Foundations for Streaming Model Transformations by Complex Event Processing. *Software and System Modeling* 17, 1 (2018), 135–162. https://doi.org/10.1007/s10270-016-0533-1

[13] Jeffrey Dean and Monika R. Henzinger. 1999. Finding Related Pages in the World Wide Web. *Comput. Netw.* 31, 11-16 (May 1999), 1467–1479.

[14] O. Etzion and P. Niblett. 2010. *Event Processing in Action.* Manning Publications.

[15] Event Processing Technical Society. 2011. *Event Processing Glossary, Version 2.0.* http://www.complexevents.com/wp-content/uploads/2011/08/EPTS_Event_Processing_Glossary_v2.pdf.

[16] Amir Hossein Ghamarian, Maarten de Mol, Arend Rensink, Eduardo Zambon, and Maria Zimakova. 2012. Modelling and analysis using GROOVE. *STTT* 14, 1 (2012), 15–40. https://doi.org/10.1007/s10009-011-0186-x

[17] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *Proc. of OSDI'12.* 17–30.

[18] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proc. OSDI'14.* 599–613.

[19] IEEE Std 1003.1-2008. 2016. *The Open Group Base Specifications. Issue 7, Sect. 4.16, Seconds Since the Epoch.*

[20] Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. 2015. *Learning Spark.* O'Reilly.

[21] Jon M. Kleinberg. 1999. Authoritative Sources in a Hyperlinked Environment. *J. ACM* 46, 5 (Sept. 1999), 604–632.

[22] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. 2009. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Mathematics* 6, 1 (2009), 29–123.

[23] David C. Luckham. 2002. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Addison-Wesley.

[24] David C. Luckham. 2012. *Event Processing for Business: Organizing the Real-Time Enterprise.* Wiley.

[25] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing. In *Proc. of SIGMOD'10.* ACM, 135–146.

[26] L. Page, S. Brin, R. Motwani, and T. Winograd. 1998. *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report. Stanford Digital Library Technologies Project. citeseer.ist.psu.edu/page98pagerank.html

[27] G. Szárnyas, B. Izsó, I. Ráth, D. Harmath, G. Bergmann, and D. Varró. 2014. IncQuery-D: A Distributed Incremental Model Query Framework in the Cloud. In *Proc. of MODELS'14 (LNCS)*, Vol. 8767. Springer, 653–669.

[28] Javier Troya, Manuel Wimmer, Loli Burgueño, and Antonio Vallecillo. 2014. Towards Approximate Model Transformations. In *Proc. AMT'14 (CEUR Workshop Proceedings)*, Vol. 1277. 44–53. http://ceur-ws.org/Vol-1277/5.pdf

[29] J. Webber, I. Robinson, and E. Eifrem. 2013. *Graph Databases.* O'Reilly Media.

[30] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, and Björn Regnell. 2012. *Experimentation in Software Engineering.* Springer.