Temporal Pattern Recognition in Graph Data Structures

Pietro Daverio
Politecnico di Milano
pietro.daverio@mail.polimi.it

Hassan Nazeer Chaudhry
Politecnico di Milano
hassannazeer.chaudhry@polimi.it

Alessandro Margara
Politecnico di Milano
alessandro.margara@polimi.it

Matteo Rossi
Politecnico di Milano
matteo.rossi@polimi.it

***Abstract*—Graph data structures model relations between entities in various domains. Graph processing systems enable scalable distributed computations over large graphs, but are limited to static scenarios in which the structure of the graph does not change. However, many applications are dynamic in nature, and this reflects to graphs that continuously evolve over time. In these contexts, understanding the evolution of graphs is key to enable timely reactions when necessary. We address this problem by proposing a new model to express temporal patterns over graph data structures. The model seamlessly integrates computations over graphs to extract relevant values, and temporal operators that define patterns of interest in the evolution of the graph. We present the syntax and semantics of our model and discuss its concrete implementation in FlowGraph, a middleware for temporal pattern recognition in large scale graphs. FlowGraph presents a level of performance that is comparable to state-of-the-art graph processing tools when processing static graphs. In the presence of temporal patterns, it can further optimize processing by avoiding complex graph computations until strictly necessary for pattern evaluation.**

***Index Terms*—temporal pattern recognition, graph data structures, distributed computations, vertex-centric computations**

I. Introduction

Many application scenarios involve relations between entities that are naturally modeled as graph-based data structures. Prominent examples are: social networks, where users are connected to each other by some "friendship" or "follower" relation; maps, where locations are connected by roads; online stores, where products are associated with customers who buy and review them; or even the World Wide Web, where pages are connected by links. In virtually all these scenarios, the graph structure evolves over time with the addition and removal of entities as well as changes in their relations. For instance, in social networks new posts are constantly added and they relate to existing ones as well as to users that read, comment, and forward them. In these contexts, common problems entail capturing and understanding the temporal evolution of the graph and its properties, thus enabling timely reactions when required. For instance, understanding the evolution of communities of users in social networks can help customize the interface to improve user experience, and also propose more suitable advertisements. Similarly, observing the relations between users and products over time in online stores can lead to better recommendations and increase profit.

978-1-6654-3902-2/21/$31.00 ©2021 IEEE

Studying the evolution of large-scale graphs over time is very challenging. On the one hand, many algorithms that extract relevant information from graphs, such as communities in social networks, are iterative in nature and computationally expensive. On the other hand, graph changes can occur frequently, so they must be analyzed with low latency to keep up with their arrival rate.

Unfortunately, existing frameworks for large-scale data processing do not meet these requirements. Graph processing systems [1] enable scalable distributed graph computations through a programming paradigm known as *think like a vertex* (TLAV) [2], introduced in 2010 with the Pregel system [3]. TLAV exploits a bulk synchronous parallel programming model, where the computation is split into supersteps (epochs): at each superstep, a vertex can perform some computation that changes its internal state and/or send messages to other vertices. This vertex-centric computing paradigm simplifies the distribution of state and computation over multiple processing nodes, but only refers to *static graphs* that do not change over time. Stream processing systems analyze dynamic data as it becomes available, to derive relevant information and enable timely reactions [4], [5]. Modern big data processing platforms such as Apache Spark Streaming [6] and Apache Flink [7] offer stream processing capabilities by implementing functional operators that transform input streams into output streams. A stream processing job is represented as a workflow of such operators, later deployed over multiple processing nodes. However, operators are designed to only store the state that is strictly needed to compute the desired results and offer limited or no support for updating large-scale data stores, as required to store graph data. In summary, despite some initial studies [8], [9], [10], the problem of defining a programming abstraction and processing framework to analyze the evolution of large-scale graphs remains open.

In this paper, we tackle this problem by introducing a novel programming model that integrates the TLAV graph processing paradigm with the temporal pattern detection capabilities of stream processing systems, and in particular of Complex Event Recognition (CER) systems [11], [12]. In our model, vertex-centric computations determine the values of properties associated with vertices and edges. Users can define temporal patterns that predicate on vertices, edges, and the values of their properties at different points in time. We present the model in detail using intuitive examples and

provide a formal definition of its semantics. We discuss the implementation of the model in FlowGraph, a middleware that enables distributed detection of temporal patterns in large-scale graphs. FlowGraph distributes the graph structure across multiple nodes that contribute to the computation and store partial results for pattern detection. The evaluation strategy of FlowGraph immediately stops the analysis of a pattern when it has no chances of being detected, which potentially avoids the execution of expensive graph computations and improves throughput. We conduct a thorough evaluation of FlowGraph under different workloads. Our results show that FlowGraph provides a level of performance on par with state-of-the-art tools when considering static graph processing. Furthermore, it can exploit temporal relations in patterns to avoid unnecessary computations and further reduce processing time when possible.

Paper outline Section II presents background information and motivates our work. Section III introduces our data and processing model, and Section IV provides their formal semantics. Section V illustrates the architecture of FlowGraph, and Section VI evaluates its performance. Section VII surveys related work and Section VIII concludes with suggestions for future research.

II. Motivations

Our work is at the intersection of two research fields: graph processing and stream processing (in particular, pattern recognition over streams of events).

Graph processing. Graph computations are known to be particularly complex, due to the inherent dependencies within graph data [13]. This problem escalates when the size of the data grows, exceeding the memory and processing capabilities of single machine solutions, and demanding for distributed processing platforms. The established approach to design scalable distributed graph computations is known as *think like a vertex* (TLAV) [2], and was introduced in 2010 with the Pregel system [3]. TLAV exploits a vertex-centric, bulk synchronous parallel programming model [14], where the computation is split into supersteps (epochs): at each superstep, a vertex can perform computations that alter its internal state and/or send messages to other vertices. The overall computation terminates when a superstep does not produce further messages. Many variants of this model exist. For instance, some systems introduce multiple phases within each superstep, as in the gather-apply-scatter (GAS) model [15], others provide subgraph-centric primitives that enable asynchronous evolution of subgraphs to some degree [16], [17], or mimic shared memory programming abstractions to ease the development of algorithms [18]. Some systems migrate data to optimize processing, for instance by reducing network traffic [19]. Despite their differences, all these approaches are designed to handle *static graphs* that do not change over time. In this context, they aim to provide processing efficiency, in terms of time to completion and use of resources, and scalability on the size of the graph. We defer to the related work Section VII the discussion of the few approaches that target dynamic graphs that evolve over time.

Stream processing. Stream processing involves the analysis of dynamic data as it becomes available, to derive relevant information and enable timely reactions. Depending on the desired output, different programming models exist [4]. One of the first and most widely adopted approaches extends the relational model to consider time-changing relations and to continuously update the results of standing queries [20]. Queries are either evaluated on a recent portion of the input data (a *window*) or on tables that are continuously modified by the incoming data. Another processing model, named Complex Event Recognition (CER) [5], looks for temporal patterns in the streams of input data. Different formalisms have been proposed for pattern specification and recognition [11], [12], ranging from regular expressions/timed automata [21], [22] to operator trees [23] and logic formulas [24], [25].

Big data processing platforms support dynamic data by providing functional operators that transform input streams into output streams. This enables distributed processing by deploying different operators on different threads or machines, and by creating multiple instances of each operator that work in parallel on different partitions of the input streams. Examples of these systems are Apache Flink [7] and Apache Spark Streaming [6]. These platforms often provide relational and CER abstractions as higher-level libraries. Operators in big data stream processing platforms can be stateless or stateful. In stateless operators, the processing of an input element only depends on the content of that element. Stateful operators retain some internal state across computations. With respect to the focus of this work, big data stream processing systems are optimized to handle large volumes of streaming data but retain state of limited size during the computation, which contrasts with the need of analyzing large-scale dynamic graphs.

Executive summary. In summary, graph processing platforms distribute the state of graphs across several processing nodes for improved performance and scalability, but only consider static graphs that do not change over time. Stream processing platforms focus on dynamic data but offer limited support for integrating large-scale state stores, as required for graph processing. As a consequence, these approaches are inadequate to assist users in detecting relevant patterns of evolution in dynamic graph data structures.

This work aims to fill this gap. We build on the CER model (recognition of temporal patterns) and augment it to support graph-shaped state and graph computations. This leads to a very expressive model that can capture complex patterns of evolution. For instance, in the context of a social network, our model could track popular users and automatically notify when some property (age, geographical region, ...) of their followers changes significantly in a short period of time. This could help, for instance, to improve advertisement campaigns over time.

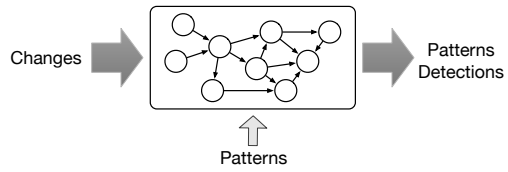


Fig. 1: FlowGraph data and processing model overview.

III. Data and Processing Model

Fig. 1 shows a conceptual overview of the FlowGraph data and processing model. FlowGraph stores a graph that continuously evolves over time according to a stream of input changes. Users install patterns that predicate on the temporal evolution of the graph, and FlowGraph notifies them whenever one of the installed patterns occur. FlowGraph adopts an event-time model [26] where input changes carry a timestamp that indicates the point in time in which they take place from the perspective of the sources. We assume that input changes are received in timestamp order. Mechanisms to cope with out-of-order arrivals of events have been discussed in the past and can be adopted to ensure this property [27]. This section introduces the language used in FlowGraph to express patterns, and Section IV formalizes its semantics.

A. Data model

The FlowGraph data model is grounded in *labeled* graphs, where each vertex and edge has associated properties (labels) in the form of key-value pairs. We also refer to the set of labels of a vertex or edge as the *state* of that vertex or edge. For instance, in social media, vertices can represent users and edges the relations among them. Labels associated with vertices can indicate properties of users, such as their name, nationality, and age, and labels associated with edges can represent the type of relation. Labels and their values can be set explicitly or derive from computations. For instance, a clustering or community detection computation can label vertices with the cluster or community they belong to.

The input stream contains time-annotated changes to the graph structure or state: addition of new vertices or edges (with their associated labels), removal of existing vertices and edges, or updates to the values of labels. We denote the collective state of all vertices and edges of a graph G after applying all the changes up to time t as the state of G at time t. Patterns consist of one or more *clauses*, which are Boolean expressions that predicate on the current and previous state of the graph. We say that a pattern is *satisfied* at time t if all its clauses evaluate to true.

B. Processing model

Pattern evaluation is triggered by input changes: whenever a change is received, FlowGraph evaluates all installed patterns and outputs a notification of detection for each and every pattern that is satisfied. Pattern clauses can refer both to the explicit values of labels in vertices and edges, or to derived values that result from computations. They can reference both the current state and the state at some previous point in time, and correlate their values. Next, we incrementally present the core language constructs to derive values from a labeled graph, and then we show how they can be combined to form clauses and patterns.

Computations. FlowGraph Supports vertex-centric computations to efficiently derive new values from the ones explicitly defined in vertices and edges. Vertex-centric computations are iterative: at each iteration each vertex updates its state and sends out messages to neighboring vertices. Developers can start a vertex-centric computation on a graph `g` using the `compute` primitive, which is parametric with respect to the following three functions. These functions are executed independently on each vertex and can augment the state of that vertex by adding more labels and iteratively updating the values of such labels.

```
init(currState: VertexStateT): VertexStateT

iterate(currState: VertexStateT, edges: Set[EdgeT],
   inMsgs: Iterator[MsgT], outMsgs: Set[(MsgT, EdgeT)]
   ): VertexStateT

end(currState: VertexStateT): VertexStateT
```

Function `init` initializes the state of each vertex before any iteration takes place. It takes in input the state of a vertex (`currState`) and outputs the initialized state. Function `iterate` defines, for each iteration, how a vertex updates its internal state and which messages it sends out. Specifically, `iterate` takes in input the current state of the vertex (`currState`), the set of outgoing edges (`edges`), and an iterator over the set of received messages (`inMsgs`). It outputs the new state of the vertex and adds outgoing messages (with the edge they need to traverse) to the `outMsgs` set. Finally, function `end` is invoked after the last iteration and returns the final state of each vertex.

To exemplify, consider an algorithm to compute the maximum value for a given label. Function `init` initializes the current maximum to the local value for each vertex. At the first iteration, each vertex sends out its local value for that label. At each subsequent iteration, a vertex updates its current view of the maximum based on the incoming messages. If its view changes after receiving a message with a larger value, then the node sends out the new value on all its outgoing edges. If the graph is connected, then eventually the algorithm converges and all the nodes agree on the same maximum value.

Vertex-centric algorithms exist for many common problems on graphs. FlowGraph includes a library of implemented algorithms as a proof of concept, which we use for testing and benchmarking. Developers can add new algorithms by implementing the `init`, `iterate`, and `end` functions.

Selection. Selection primitives isolate a subgraph based on the values of labels in vertices (`selectV`) or edges (`selectE`). Specifically, a `select` primitive takes in input a predicate, that is, a function that evaluates the state of vertices or edges and returns a Boolean value. It retains the vertices or edges for which the function evaluates to true. The `selectV` primitive also retains all and only the edges that connect selected

vertices. The selectE primitive also retains all vertices that are sources or destinations of selected edges.

For instance, consider the computation of the shortest paths from a given vertex: selectV might isolate the subgraph containing all the vertices whose shortest path is below a given threshold, together with the edges that compose the path, as exemplified in the snippet below. First, we compute the shortest path tree on the graph starting from vertex v, using the init(), iterate() and end() functions defined for the shortest path algorithm. Then, we select all vertices and edges having a value lower than 10 for label distance. This label links the shortest path computation and the subsequent selection: the computation assigns the label to each vertex, and the selection evaluates the label to identify a subgraph. FlowGraph lets user customize the name of the labels used in computations, such that the results of multiple computations do not conflict.

```
graph.compute(ShortestPath.fromVertex(v),
      ShortestPath.iterate(), ShortestPath.end())
     .selectV(distance < 10)
```

All the primitives that work on graphs can be applied to selected subgraphs. For instance, one could start a new computation that considers only the selected subgraph.

Value extraction. Value extraction primitives let users refer to values of labels inside vertices (extractV) or edges (extractE). The primitives take in input a list of labels l and return a set of lists of values. Each list in the result set contains the values associated with the labels in l for one vertex (in the case of extractV) or edge (in the case of extractE) in the graph. Each vertex and edge has an implicit and immutable label id, representing a unique identifier that the system associates with that vertex or edge. FlowGraph always includes id in the list of extracted labels, so that users always obtain the identity of the vertex or edge as part of the extracted values.

To examplify, the following snippet extracts all the distances assigned to the vertices of the graph identified in the previous example. Specifically, the extractV primitive takes in input a list consisting of a single label (distance) and returns, for each vertex in the graph, the id of that vertex and the value associated with that label.

```
graph.compute(ShortestPath.fromVertex(v),
      ShortestPath.iterate(), ShortestPath.end())
   .selectV(distance < 10)
   .extractV(distance)
```

Functional operators. In line with modern big data processing frameworks, FlowGraph provides a library of functional operators to derive new values starting from extracted ones. Functional operators include filter, which filters the values according to a predicate, map and flatMap, which transform each input element into one or more output elements according to a user-defined function. An important class of functional operators are reductions, which aggregate all input values into a single output result. FlowGraph provides common arithmetic reductions such as maximum, minimum, and average out-of-the-box. For instance, the following code snippet computes the average distance from values extracted in the previous example.

```
graph.compute(ShortestPath.fromVertex(v),
      ShortestPath.iterate(), ShortestPath.end())
   .selectV(distance < 10)
   .extractV(distance).avg(distance)
```

Definition of subgraphs. FlowGraph provides primitives to identify subgraphs where vertices (subgraphByV) or edges (subgraphByE) share common values for one or more labels. Subsequent operations are then applied to each and every subgraph independently. Consider for instance the following code snippet. It first runs a community detection algorithm that associates a community label with each and every vertex. Then, it defines subgraphs having vertices that share the same value for the community label. Finally, it extracts the set of vertices for each of these subgraphs, and computes the cardinality of each set.

```
graph.compute(CommunityDetection.init(),
              CommunityDetection.iterate(),
              CommunityDetection.end())
   .subgraphByV(community)
   .extractV().count()
```

When using subgraphByV, a subgraph contains all and only the edges having both the source and the destination vertices in that subgraph. When using subgraphByE, a subgraph contains all the vertices that are either source or destination for an edge in that group.

Variables. Using the emit primitive, FlowGraph defines variables to bind values in different parts of a pattern. Variables can refer to graphs or values extracted from computations on graphs. For instance, the following code snippet counts the number of people older than 20 from the largest community (or communities).

```
graph.compute(CommunityDetection.init(),
              CommunityDetection.iterate(),
              CommunityDetection.end())
   .subgraphByV(community).emit(communityGraphs)
   .extractV().count().emit(communitySize)
   .max().emit(maxSize)

communityGraphs.select(communitySize == maxSize)
   .extractV(id, age).selectV(age > 20).count()
```

The first part of the pattern computes communities and groups vertices according the their value for the community label. It associates such groups (graphs) with a variable communityGraphs. Then, it computes the number of vertices in each graph and stores it into a communitySize variable: this associates a different value with communitySize for each group. Finally, the first part of the pattern computes the maximum size of communities and assigns it to a variable maxSize. This example illustrates the flexibility of variables, which can refer to graphs (as in the case of communityGraphs), multiple values (as in the case of communitySize), or a single aggregated value (as in the case of maxSize).

The second part of the pattern starts from the graphs in `communityGraphs` and selects the one (or ones) having maximum size. This selection refers to the `communitySize` and `maxSize` variables previously emitted. Finally, the pattern selects and counts the vertices having a value greater than 20 for label `age`.

Temporal operators. To predicate on the temporal evolution of a graph, users can refer to values at different times. Specifically, users can refer to the value at a specific point in time (relative to the evaluation time), or to all values in a window of time. For instance, the following snippet refers to the size of the graph (number of vertices) 10 seconds before the time of evaluation. The `before` primitive takes in input a temporal value `t` with its time unit, and returns the value of a label or computation as if it was performed `t` time units before the current time.

```
graph.extractV().count()
   .before(10, TimeUnit.SECONDS)
```

Similarly, the following snippet computes the maximum size of the graph in the last 10 seconds. The `window` operator takes in input a temporal value `t` with its time unit, and returns the list of values that a label or computation assumed in a time window starting `t` time units ago (included) and ending now (excluded), one for each point in time in which the graph changed (that is, a change was received from the input stream).

```
graph.extractV().count()
   .window(10, TimeUnit.SECONDS).max()
```

Pattern clauses. Patterns consist of multiple clauses, each of them introducing a constraint over some value derived from the labeled graph at the current time or at some previous point in time. Clauses are defined with the `evaluate` primitive, which takes in input a predicate and applies it to a value. For instance, the following snippet defines a clause that is satisfied whenever (at least) one community larger than 20 is detected.

```
graph.compute(CommunityDetection.init(),
              CommunityDetection.iterate(),
              CommunityDetection.end())
   .subgraphByV(community).extractV()
   .count().evaluate(x -> x > 20)
```

IV. Formal semantics

This section provides the formal semantics for the data and processing model presented above.

A. Data model

We model temporal evolution by considering different graphs, each of them representing the state of a time-evolving graph at a given point in time. Accordingly, we model a graph as a 3-tuple $G = (V_G, E_G, t_G)$, where V_G is the set of vertices, E_G is the set of edges, and t_G is a timestamp. A vertex $v \in V_G$ is a pair $v = (id, \ell)$, where id is a unique identifier of the vertex and ℓ is the state of that vertex, that is, a set of labels (key-value pairs) associated with that vertex. An edge $e \in E_G$ is a 4-tuple $e = (id, v_s, v_d, \ell)$, where id is a unique identifier of the edge, v_s and v_d are the source and destination vertices, and ℓ is a set of labels (key-value pairs) associated with that edge. We denote $v.\ell$ (respectively, $e.\ell$) the set of labels associated with vertex v (edge e), and $v.\ell.k$ ($e.\ell.k$) the value associated with key k in vertex v (edge e). A unique identifier for a vertex v (edge e) is stored in $v.\ell.id$ ($e.\ell.id$).

We model the input stream S as a (possibly unbounded) sequence of timestamped notifications (N_i, t_i), where N_i is a set of changes to the structure or to the state of the graph (addition or removal of vertices or edges, or changes in the value of labels), and t_i is a timestamp that indicates the point in time when the changes occur.
We assume timestamps to be monotonically increasing:

$$\forall_{i,j \in \mathbb{N}}\ ((N_i, t_i) \in S \wedge (N_j, t_j) \in S \wedge i > j) \rightarrow t_i > t_j$$

Assume that FlowGraph stores a graph $G = (V_G, E_G, t_G)$. Receiving input element (N_i, t_i) leads to a new graph $G' = (V_{G'}, E_{G'}, t_i)$, where $V_{G'}$ and $E_{G'}$ are computed from V_G and E_G by applying all the changes in N_i. Thus, each input element at time t results in a new graph at time t. When a pattern refers to a graph at time t', it considers the graph with the largest timestamp such that $t \leq t'$ holds, meaning that the graph was defined by all input elements up to time t and there are no other input elements between t and t'.

B. Processing model

A pattern $p \in P$ is a conjunction of clauses $p = c_1^p \wedge \cdots \wedge c_n^p$. Each clause predicates on a value at some point in time. A pattern evaluation is triggered by the arrival of an input element. The pattern is satisfied if all its clauses evaluate to true, in which case FlowGraph emits a notification of detection for that pattern. Values in clauses can be fully specified or they can depend on a set of one or more variables $V = v_1, \ldots, v_n$. In the latter case, the pattern is satisfied if, and only if, there is at least one assignment of values $v'_1, \ldots, v'_n$ for the variables in V that satisfy the pattern.

Computations. A computation takes place at some point in time t and updates the set of labels without changing the structure of the graph (that is, without adding or removing vertices or edges) and its timestamp. We formalize a computation as a function $comp : \overline{G} \rightarrow \overline{G}$, where $\overline{G}$ is the set of all possible graphs, subject to the following constraints: if $G' = comp(G)$ then time, vertices and edges (identifiers) remain the same.

$$t_{G'} = t_G$$

$$(\forall_{v \in V_G}\ \exists_{v' \in V_{G'}}\ v.\ell.id = v'.\ell.id) \wedge (\forall_{v' \in V_{G'}}\ \exists_{v \in V_G}\ v.\ell.id = v'.\ell.id)$$

$$(\forall_{e \in E_G}\ \exists_{e' \in E_{G'}}\ e.\ell.id = e'.\ell.id) \wedge (\forall_{e' \in E_{G'}}\ \exists_{e \in E_G}\ e.\ell.id = e'.\ell.id)$$

Labels (other than id) can be added or modified according to the specific semantics of the computation, which is out of the scope of this formalization.

Selection. We model a label predicate as a function p that takes in input a set of labels and returns a Boolean value $p : \mathcal{P}(L) \rightarrow bool$, where L is the set of all possible labels and $\mathcal{P}(L)$ is its power set. Let us denote P the set of all possible predicates. We now model the `selectV` operator, being `selectE` similar. We model selection as a function $sel : \overline{G} \times P \rightarrow \overline{G}$ that takes in input a graph G and a

label predicate p and returns a new graph G', subject to the following constraints: G' has the same time as G, contains all and only the vertices that satisfy the selection, and all and only the edges that connect such vertices.

$$t_{G'} = t_G$$

$$\forall_{v' \in V_{G'}} \ (v' \in V_G) \ \wedge \ \forall_{v \in V_G} \ (v \in V_{G'} \leftrightarrow p(v.\ell))$$

$$\forall_{e' \in E_{G'}} \ (e' \in E_G) \ \wedge \ \forall_{e \in E_G} \ (e \in E_{G'} \leftrightarrow (p(e.v_s.\ell) \wedge p(e.v_d.\ell))$$

Value extraction. For value extraction we refer to the `extractV` predicate, being `extractE` similar. Let KL denote the set of keys in labels and VL the set of values in labels. We model values extraction as a function $extract : \overline{G} \times \mathcal{P}(KL) \rightarrow \mathcal{P}(VL)$ that takes in input a graph $G \in \overline{G}$ and a set of keys $keys = \{k_1, \ldots, k_n\} \in KL$, and returns a set of tuples $(v_{id}, v_1, \ldots, v_n)$, one for each vertex v in G, where v_{id} is the unique identifier of vertex v and $v_i \in VL$ is the value associated with k_i in vertex v.

$extract(G, keys)$ contains as many elements as the number of vertices in G.

$$|extract(G, keys)| = |V_G|$$

Each element in $extract(G, keys)$ contains the values of the labels of one vertex in G.

$$(v_{id}, v_1, \ldots, v_n) \quad \in \quad extract(G, keys) \quad \leftrightarrow$$
$$\exists_{v \in V_G} \ \left(v_{id} = v.\ell.id \wedge \forall_{k_i \in keys} v_i = v.\ell.k_i\right)$$

Functional operators. Functional operators compute an output dataset starting from an input dataset. The semantics of the computation is provided via user-defined functions and is outside the scope of this formalization.

Definition of subgraphs. We provide the semantics of `subgraphByV`, being the definition of `subgraphByE` similar. In its simplest form, `subgraphByV` computes a set of graphs GS starting from a single graph G. We start to formalize this case, and then discuss how subgraph operators can be applied recursively. We model `subgraphByV` as a function $subgraphByV : \overline{G} \times \mathcal{P}(KL) \rightarrow \mathcal{P}(\overline{G})$ that takes in input a graph $G \in \overline{G}$ and a set of label keys $keys \in \mathcal{P}(KL)$ and returns a set of graphs $GS \in \mathcal{P}(\overline{G})$, subject to the following constraints.

Each graph G' in GS has the same time as G.

$$\forall_{G' \in GS} \ t_{G'} = t_G$$

Each graph G' in GS can only contain vertices and edges that are in G.

$$\forall_{G' \in GS} \ (v \in V_{G'} \rightarrow v \in V_G) \ \wedge \ (e \in E_{G'} \rightarrow e \in E_G)$$

All the vertices in a graph G' in GS contain the same values for all labels in $keys$.

$$\forall_{G' \in GS} \ \forall_{k \in keys} \ \forall_{v, v' \in V_{G'}} \ (k, val) \in v.\ell \rightarrow (k, val) \in v'.\ell$$

In the following definitions, k_G denotes the value that all vertices in a graph $G \in GS$ share for the label with key k. Different graphs G' and G'' in GS contain different values for at least one key.

$$\forall_{G', G'' \in GS} \ G' \neq G'' \ \rightarrow \ (\exists_{k \in keys} \ k_{G'} \neq k_{G''})$$

There is one graph G' in the result set for each distinct set of key values.

$$\forall_{v \in V_G} \ \forall_{k \in keys} \ \exists_{G' \in GS} \ k_{G'} = v.\ell.k$$

An edge is contained in a graph G' if and only if both its source and its destination vertices are.

$$\forall_{G' \in GS} \ \forall_{e \in E_G} \ e \in E_{G'} \leftrightarrow (e.v_s \in V_{G'} \wedge e.v_d \in V_{G'})$$

In the general case, `subgraphByV` can be applied repeatedly, thus leading to groups of groups of graphs and so on. We model this case by generalizing the definition above. First, we introduce the concept of group $Gr \in \overline{Gr}$, which is either a graph $G \in \overline{G}$, or a set of groups. Then, we redefine `subgraphByV` to work on groups. Let us define a predicate $isGraph : \overline{Gr} \rightarrow bool$ that takes in input a group and returns true if the group is a single graph. We also define a function $nestLev : \overline{Gr} \rightarrow \mathbb{N}$ that defines, for each group, its *nesting level*, where the nesting level of an atomic graph is 0, while that of a group is well-defined and equal to $n > 0$ if, and only if, each element of the group has the same nesting level equal to $n - 1$:

$$\forall_{Gr \in \overline{Gr}} \ ((isGraph(Gr) \rightarrow nestLev(Gr) = 0) \wedge (\neg isGraph(Gr) \rightarrow (nestLev(Gr) = n \leftrightarrow \forall_{G' \in Gr}(nestLev(G') = n - 1)))$$

The general definition of `subgraphByV` takes in input a group Gr and a set of label keys $keys$ and returns a group Gr'. We require that the nesting level of Gr be well defined:

$$subgraphByV(Gr, keys) = Gr' \rightarrow \exists_{n \in \mathbb{N}}(nestLev(Gr) = n)$$

If the group consists of a graph, then it returns a set of graphs as defined above (we do not repeat this base case for the sake of space). If the group is a set, it calls recursively `subgraphByV` on each and every element Gr'' of the set.

$$\neg isGraph(Gr) \rightarrow Gr' = \{\Gamma | \exists_{Gr'' \in Gr} subgraphByV(Gr'', keys) = \Gamma\}$$

Temporal operators. Temporal operators let users refer to graphs at different points in time. This does not change the semantics of other language constructs, but only the graph these constructs are applied to.

We model the `before` operator as a bef function that takes in input a time point $t \in T$ and returns the most recent graph at time t: $bef : T \rightarrow \overline{G}$. Recall that S is the stream of input changes. $bef(t)$ returns the value at the point in time t' when the last notification N' before t was received from the input stream S.

$$bef(t) = G \leftrightarrow \exists_{(N', t') \in S} t_G = t' \wedge t' \leq t \wedge \nexists_{(N'', t'') \in S} t' < t'' \leq t$$

We model the `window` operator as a win function that takes in input a time point $t \in T$ and returns the set of all graphs between t and the current time (t_{now}): $win : T \rightarrow \mathcal{P}(\overline{G})$.

$$win(t) \ = \ \{G \mid t \leq t_G \leq t_{now} \ \wedge \ \exists (N, t_G) \in S\}$$

Notice that our language enables temporal operators to be applied not only to graphs, but more generically to values derived from graphs at different points in time. This is equivalent to first applying temporal operators to identify graphs at some point in time, and then deriving some values from them. So, the above definitions are sufficient to express all the temporal constructs in our model.

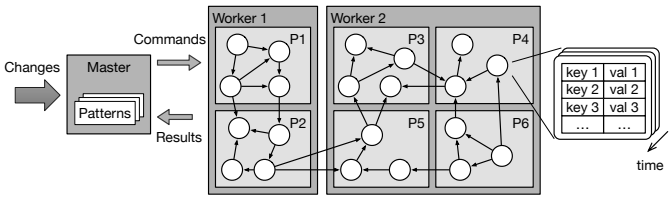


Fig. 2: Architecture of FlowGraph.

V. Middleware Implementation

FlowGraph is an open source project written in Java on top of the Akka actor system[1]. Fig. 2 depicts the architecture of FlowGraph. It comprises a master node that coordinates many worker nodes. Clients connect to the master node and submit the patterns of interest together with the code of any user-defined computation. We implemented a parser of patterns using the ANTLR parser generator[2].

Graph vertices and edges are partitioned across worker nodes. Workers (dark grey boxes in Fig. 2) are processes, potentially running on different machines. Each partition (light grey box in Fig. 2) within a worker is handled by an actor. Workers can get a different number of partitions, depending on the computational and memory resources of the machine where they are deployed. For instance, Fig. 2 shows a deployment with two workers and six partitions. `Worker 1` manages partitions `P1` and `P2`, while `Worker 2` manages partitions `P3`, `P4`, `P5`, and `P6`.

Each vertex obtains a unique identifier upon creation and is assigned to a partition based on a hash of its identifier. Edges are assigned to the same partition as their source vertex. Optimized partitioning of vertices based on graph topology as well as dynamic repartitioning upon change are currently outside the scope of this work, but we plan to integrate both aspects by building on state-of-the-art approaches [28], [29].

Workers store the state of the graph in main memory for improved performance. As shown in Fig. 2, workers store the state of their portion of the graph in a multi-version key-value table: it contains the labels of each vertex and each edge at multiple points in time, indexed by time and key. Old versions are deleted from the stores as soon as they cannot influence the detection of any pattern anymore. Their time of validity is determined by statically analyzing the patterns when they are deployed into the system.

Execution model. The input stream of changes is handled by the master that redirects each change to the partition responsible for it. The master also governs the execution of the various computational steps that are necessary to evaluate a pattern. Specifically, the master issues commands to the workers indicating the type of primitive they need to execute. Data remains local to workers that return to the master the minimum information that is necessary to evaluate the pattern. Patterns are evaluated sequentially, one after the other.

Computations. Workers execute computations using a vertex-centric paradigm, with the master acting as a synchronization point between epochs. As discussed in Section III, computations are parametric with respect to three functions that specify how vertices initialize and update the state of the computation at each iteration, and how they exchange information. When executed on a vertex v, the initialization, iteration, and termination functions are allowed to modify the current version of the key-value store associated with v by adding new labels and iteratively updating their values.

Communication is implemented in a hierarchical way. Messages that are local to a worker are exchanged through shared memory. Messages across workers are serialized and sent through the network. Workers exchange messages directly without passing through the master. At the end of each iteration, each worker notifies the number of messages generated during the iteration by all vertices in its partitions. The master collects notifications from each and every worker, and starts a new iteration only if some message has been generated.

Selection. When the master commands a selection, each worker independently performs the operation on all the vertices (or edges) in its partitions. A single inter-worker communication step is necessary in the case of edges that cross the boundaries of partitions, to determine whether the edge and its connected vertices are part of the selection. Each worker then locally flags selected vertices and edges, and considers only flagged entities in subsequent operations.

Values extraction. Extraction also takes place independently on each worker, which simply converts each vertex (or edge) into a set of values.

Functional operators. Functional operators transform extracted values, following the same approach as distributed stream processing frameworks. Actors within workers operate in parallel on the partitions they are responsible for. Several operators such as `filter`, `map`, or `flatmap` simply convert each element in the input dataset into one or more elements in the output dataset, without requiring any communication between workers. Other operators, however, require exchanging data. For instance, reduction operators compute a single value from a dataset. FlowGraph implements several reduction operators. Whenever possibile, the process takes place hierarchically by first combining values within a partition, and then reducing the values across partitions. The unique result of a reduction is broadcast to all workers, as it might be used in subsequent evaluations of the pattern.

Definition of subgraphs. FlowGraph implements subgraph primitives as local operations within each partition. It does not move vertices or edges across partitions, but simply annotates each vertex and edge with an identifier of the subgraph (or subgraphs) it belongs to. Any subsequent operation that is performed within a subgraph will take this identifier into account. For instance, a computation will exchange messages only across vertices that are part of the same subgraph.

Temporal operators. In the presence of temporal operators, the master computes the point in time t (or time window w) to be considered for the subsequent commands, and communicates it to the workers. Workers follow the same approach

[1]https://akka.io/docs/

[2]https://www.antlr.org

discussed above, but refer to the version of the key-value store valid at time t (or within the time window w).

Variables and evaluation. Evaluation of pattern clauses also takes place in workers. Indeed, workers store any value that derives from graph computations, functional, and temporal transformations, so they can autonomously evaluate a predicate on a value and return the result to the master. In the case a clause depends on a variable previously computed during the evaluation of a pattern, the master specifies which value the variable refers to.

VI. Evaluation

Our evaluation has several goals: (i) study the absolute performance and scalability of FlowGraph in executing vertex-centric computations; (ii) compare FlowGraph against state-of-the-art solutions for distributed vertex-centric computations; (iii) study how the constructs offered by our pattern definition language affect performance, scalability, and use of resources.

To answer question (i), we execute the page rank vertex-centric algorithm while increasing the size of the graph (number of vertices and number of edges) and the available processing resources. To answer question (ii), we compare our system with GraphX [30], a state-of-the-art library for graph processing in distributed environments. GraphX builds on Apache Spark [6] and is widely adopted for its efficiency and scalability. To answer question (iii), we perform detailed microbenchmarking and isolate the impact of various pattern constructs on the performance of FlowGraph.

A. Experiment setup

We now present the setup of our evaluation in terms of processing infrastructure, dataset, parameters that we control, and values that we measure.

Processing infrastructure. To enable reproducibility of results, we execute all our experiments on a public cloud infrastructure. We deploy FlowGraph on *m5.2xlarge* EC2 instances of Amazon AWS. Each instance is powered by 8 vCPU (4 cores, 2 threads per core) running on *Intel Xeon® Platinum 8175* processors at up to 3.1 GHz, backed by 32 GB of memory and up to 10 Gbps of network bandwidth.

Dataset. To make sure that we measure the performance of FlowGraph when it is in a steady state, we load a graph into FlowGraph before starting any evaluation. The graph we load is directed, fully connected, with an average out-degree of 2. Each vertex has a label `label` with a numeric value uniformly distributed between 1 and 4, included. In the remainder, we will refer to the number of vertices in the graph as its *size*. We inject one input (graph change) at a time and we average our measurements over at least 10 inputs (100 when considering graphs smaller than one million vertices). Unless otherwise specified, each input modifies the state of a vertex, but not the graph topology. We use the page rank algorithm as vertex-centric computation. To ensure that the results across several executions are comparable, we consider a fixed number of iterations without checking and stopping the iterative process in the case of convergence.

Parameter	Default
Size of the graph	1M
Average out-degree	2
Number of instances (VMs)	4
Number of workers per instance	8
Number of partitions	32
Computation	Page rank (10 iterations)

TABLE I: Parameters used in the evaluation.

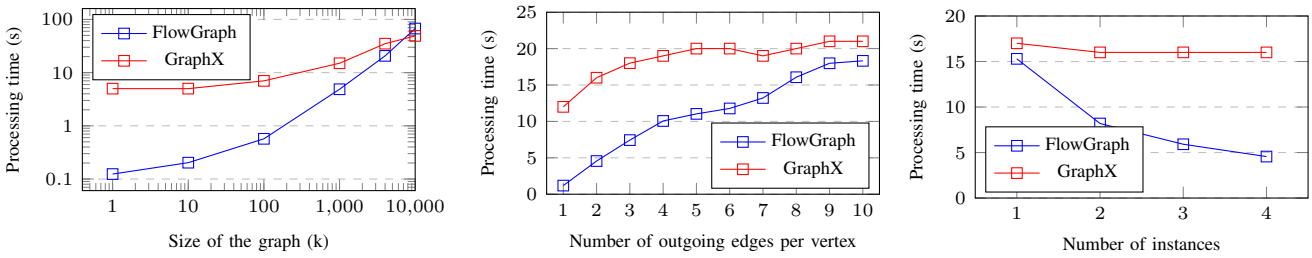
Measured values. As FlowGraph is designed to handle dynamic data, we are primarily interested in understanding how fast it can process input data (graph changes). Accordingly, we measure the *average processing time* per input element as the difference between (i) the point in time when an input element starts to be actively processed by the middleware, and (ii) the point in time when the middleware ends processing that element, after producing all the pattern detection results, if any. The average processing time represents the response time of the system when not overloaded, that is, when there are no input elements waiting to be processed in input queues. The inverse of the average processing time also gives a good estimate of the number of elements that FlowGraph can process in a unit of time, that is, its maximum *sustainable input throughput* [31]. In addition, when relevant, we also measure the memory utilization of FlowGraph as the amount of memory used by one JVM process at a defined point of execution.

Parameters. To evaluate FlowGraph under heterogeneous operating conditions, we consider several parameters that affect its performance. We summarize them in Table I, showing their default value when not differently specified.

B. Vertex-centric computations

First, we focus on vertex-centric computations and we measure 1) the average processing time of FlowGraph when increasing the number of vertices and the number of edges in the graph; 2) the scalability of FlowGraph when increasing the number of available instances. Since we focus on vertex-centric computations on a static graph, we compare the performance of FlowGraph with GraphX. During these experiments we recompute the page rank algorithms every time we receive an input element. We consider a fixed number of 10 iterations. For GraphX, we use Apache Spark 3.0.0 and the page rank implementation provided by the library.

Fig. 3a compares the average processing time of FlowGraph and GraphX while increasing the size of the graph from 1k vertices to 10M vertices. In absolute terms, FlowGraph performs the computation under 0.6s for a graph of 100k vertices and under 5s for a graph of 1M vertices. The processing time increases more than linearly when moving from 1k to 100k vertices, but then starts growing linearly. We believe this is because the processing time is dominated by a fixed message communication overhead with graphs of small sizes. The same trend appears in GraphX, although the overhead of the Spark platform is larger. In fact, FlowGraph outperforms GraphX with up to 10M vertices, and remains comparable with 10M vertices, despite GraphX being a mature commercial product optimized for distributed processing. While the focus of our



(a) Number of vertices

(b) Number of edges

(c) Number of instances

Fig. 3: Page rank computation

research is temporal pattern recognition, this proves the efficiency of our prototype implementation in distributed vertex-centric computations. Fig. 3b compares the processing time of FlowGraph and GraphX when increasing the number of outgoing edges for each vertex. Both systems exhibit a similar trend, with a sub-linear increase in processing time: indeed, the number of vertices remains constant but they exchange more messages at each iteration. Interestingly, FlowGraph outperforms GraphX in all the scenarios we tested. Finally, Fig. 3c shows how FlowGraph scales when increasing the number of instances. Despite inter-instance communication, FlowGraph clearly takes advantage of the added processing resources. Remarkably, it obtains a speedup of 3.3× when moving from 1 to 4 instances. In comparison, GraphX performs comparably with one instance, but presents marginal improvements when moving from 1 to 4 instances. This is probably due to the higher overhead of the Spark platform already discussed Fig. 3a and in previous literature [32]. We also observed the same trend with larger graphs (up to 10M vertices, not reported for brevity).

C. Pattern detection

We now evaluate pattern detection constructs.

Selection. To evaluate the performance of selection, we measure the average processing time to select vertices with a specific value for label `label`. Recall that `label` gets a uniform value between 1 and 4, hence we select 25% of the vertices. Fig. 4a shows the average processing time when the size of the graph increases. Since selection requires evaluating a condition on each and every vertex, the average processing time increases linearly with the size of the graph. In absolute terms FlowGraph can handle selection over 5M vertices in about 0.67s, and over 10M vertices in about 1.6s.

Definition of subgraphs. Fig. 4b shows how the performance of FlowGraph changes when considering subgraph definition and selection. We compare three different patterns: `no group` performs a computation (10 iterations of page rank) on the entire graph, as in the previous section. `group` groups vertices according to their value of the label `label` and then performs the computation on each group. `group select` groups vertices and then selects only one group. Recall that each group contains about 25% of the vertices. The definition of subgraphs (`subgraphByV` operator) requires about 1.2s on a graph of 1M vertices. However, performing the computation on smaller subgraphs rather than the entire graph reduces the `compute` time from about 4.5s to about 0.9s. The `select` operator requires about 0.2s to run, but it further reduces the `compute` time to about 0.1s, since page rank can converge in fewer iterations in the selected graph. These results confirm that FlowGraph can effectively define subgraphs and operate on them, and this has the potential to speed up computations with respect to considering the whole graph, as we observed in the case of page rank.

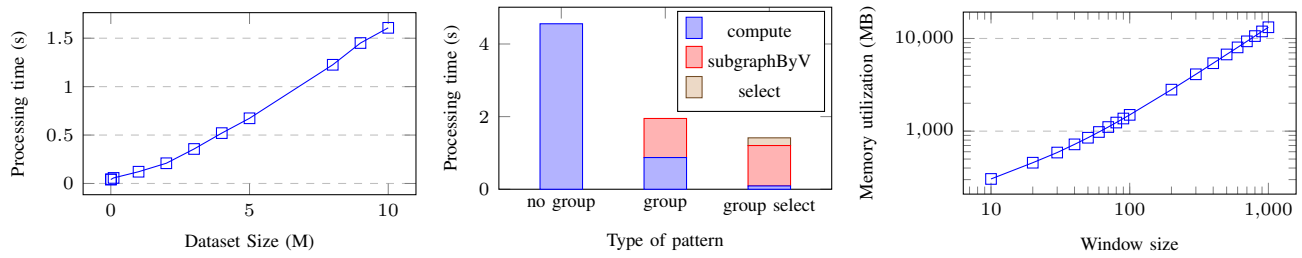
Windowed evaluations. We now consider a temporal pattern that evaluates graphs over a window of time. The processing overhead for the evaluation when increasing the size of the window is negligible, so the average processing time remains almost constant. However, the presence of a window requires storing different versions of the graph. Fig. 4c shows the average memory utilization per instance when increasing the size of the window. As expected, the memory utilization grows linearly. In terms of absolute values, the memory utilization of each instance remains below 14 GB even when considering a window size of 1000s.

Temporal sequences. We now evaluate the performance of FlowGraph in detecting temporal sequences, and we show how they can be used to optimize the average processing time by triggering computations only when certain conditions hold. To do so, we consider the following pattern

```
graph.compute(OutDegree.init(), OutDegree.iterate(),
   OutDegree.end()).extractV(numOutEdges).count()
     .before(10, TimeUnit.MINUTES).emit(previousSize)
graph.compute(OutDegree.init(), OutDegree.iterate(),
   OutDegree.end()).extractV(numOutEdges).count()
     .evaluate(size -> size - previousSize > 10)

graph.compute(PageRank.init(), PageRank.iterate(),
              PageRank.end())
```

The first clause of the pattern computes the out-degree (number of outgoing edges) of each vertex. It compares the total number of edges at the time of evaluation with the total number of edges 10 minutes before the time of evaluation (stored in variable `previousSize`). The clause is satisfied only if the difference in size is greater than 10. Finally, the pattern computes page rank. We test the pattern on a stream of changes, where each change adds a new edge to the graph. Under these circumstances, FlowGraph avoids computing the



(a) Selection (b) Subgraph definition (c) Windows. Memory per machine.

Fig. 4: Pattern definition

	OutDegree comput.	**OutDegree eval.**	**PageRank comput.**	**Total**
Page rank	0s	0s	4.56s	4.56s
Temporal sequence	0.10s	0.10s	0.46s	0.66s

TABLE II: Temporal sequence: computing page rank always vs only when the number of edges increases by 10 in 10 min.

second clause when the first one evaluates to false. This results in avoiding an expensive page rank computation when the number of edges has not increased significantly in the last 10 minutes. Table II compares the average processing time when evaluating the page rank clause for each input element (first line) and when using the above pattern (second line). We consider an input of 300 changes such that the page rank computation is executed only 10% of the times. As Table II shows, the computation of the out degree and the evaluation of its difference over time affects performance only marginally (0.2s on average), but the time spent to compute page rank decreases by almost 10 times, since page rank is only evaluated on 10% of the input. As a consequence the average processing time decreases from 4.56s to 0.66s. This proves that (i) FlowGraph computes temporal sequences efficiently, and (ii) using temporal constraints can avoid complex computations when they are not needed, significantly decreasing the average processing time.

VII. Related Work

Section II already presented vertex-centric computations and logic-based temporal pattern recognition, which are the building abstractions for FlowGraph. This section reviews approaches that deal with dynamic graphs.

A recent survey on dynamic graph analysis [33] classifies existing work in the area in two categories: *maintenance* methods, which maintain (possibly with incremental algorithms) the results of a computation as the graph evolves, and *evolution analysis* methods, which aim to *quantify* and *understand* the changes that occurred in the underlying graph. Our work fits into the second class, although it can benefit from efficient maintenance methods to update the results of computations used in patterns. Evolution analysis methods focus on specific problems such as community emergence and evolution [34] or shortest path distance evolution [35]. Our work is more general as it can integrate the results of multiple computations within a pattern, although it does not focus on the optimization of any specific algorithm. Only few systems implement evolution analysis problems in centralized or distributed settings, in batch or in near-real-time/streaming fashion [36], [37], [38], [10]. These systems are the most closely related to our proposal. However, to the best of our knowledge, our work is the first to provide a formal specification of temporal patterns over dynamic graphs. Song et al. [39] introduce an algorithm to detect patterns over dynamic graphs. Differently from our proposal, they look for structural patterns (a problem known as *subgraph pattern matching* [40]) and extend it to capture a strict partial order over time when the vertices and edges that form the subgraph are added. Although our model can support subgraph pattern matching through computations, we plan to include ad-hoc constructs and efficient evalutation algorithms for this problem in the future.

Graphs are also at the heart of knowledge representation in many domains. For instance, in semantic Web, queries to a knowledge base take the form of subgraph pattern matching, as in the standard SPARQL language [41]. Several works extended SPARQL to reason on streaming data [42], [43], [44]. Some recent work proposed a logic framework to express and recognize temporal sequences of subgraph patterns [45], [46]. We believe that the distributed architecture presented in this paper can be beneficial in this area of application.

Graph databases enable storing and querying graph data [47]. Temporal graph databases exist [8], but do not address the detection of temporal patterns as we do.

VIII. Conclusions

This paper proposes a novel programming model to express patterns of changes in temporal graph data structures. The model combines vertex-centric computations to extract relevant information from graphs with temporal operators to define patterns of interest that predicate on the evolution of the graph. We implement the model in the FlowGraph distributed platform. Our evaluation shows that the performance of FlowGraph is comparable to state-of-the-art distributed frameworks for static computations, and enables further optimizations in presence of temporal patterns. Our plans for future work include: (i) extend the library of vertex-centric algorithms, also considering incremental computations; (ii) introduce operators for subgraph pattern matching [40]; (iii) study vertex migration approaches to reduce the cost of computation [28], [19]; (iv) investigate pattern rewriting techniques [48] to optimize evaluation.

REFERENCES

[1] V. Kalavri, V. Vlassov, and S. Haridi, "High-level programming abstractions for distributed graph processing," *Trans. on Knowledge and Data Engin.*, vol. 30, no. 2, 2018.

[2] R. R. McCune, T. Weninger, and G. Madey, "Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing," *ACM Comp. Surveys*, vol. 48, no. 2, 2015.

[3] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Int. Conf. on Management of Data*, ser. SIGMOD '10. ACM, 2010.

[4] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Comp. Surveys*, vol. 44, no. 3, 2012.

[5] O. Etzion and P. Niblett, *Event Processing in Action*. Manning, 2010.

[6] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Symp. on Operating Sys. Principles*, ser. SOSP '13. ACM, 2013.

[7] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *IEEE Data Engin. Bullettin*, vol. 38, no. 4, 2015.

[8] Y. Miao, W. Han, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, E. Chen, and W. Chen, "Immortalgraph: A system for storage and analysis of temporal graphs," *Trans. on Storage*, vol. 11, no. 3, 2015.

[9] A. P. Iyer, L. E. Li, T. Das, and I. Stoica, "Time-evolving graph processing at scale," in *Int. Workshop on Graph Data Management Experiences and Sys.*, ser. GRADES '16. ACM, 2016.

[10] B. Erb, D. Meissner, J. Pietron, and F. Kargl, "Chronograph: A distributed processing platform for online and batch computations on event-sourced graphs," in *Int. Conf. on Distributed and Event-based Sys.*, ser. DEBS '17. ACM, 2017.

[11] A. Artikis, A. Margara, M. Ugarte, S. Vansummeren, and M. Weidlich, "Complex event recognition languages: Tutorial," in *Int. Conf. on Distributed and Event-based Sys.*, ser. DEBS '17. ACM, 2017.

[12] N. Giatrakos, E. Alevizos, A. Artikis, A. Deligiannakis, and M. Garofalakis, "Complex event recognition in the big data era: a survey," *VLDB*, 2019.

[13] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in Parallel Graph Processing," *Parallel Proc. Letters*, vol. 17, no. 01, 2007.

[14] A. V. Gerbessiotis and L. G. Valiant, "Direct bulk-synchronous parallel algorithms," *Parallel and Distributed Computing*, vol. 22, no. 2, 1994.

[15] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Conf. on Operating Sys. Design and Impl.*, ser. OSDI'12. USENIX, 2012.

[16] Y. Simmhan, A. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. Raghavendra, and V. Prasanna, "Goffish: A sub-graph centric framework for large-scale graph analytics," in *Euro-Par Parallel Proc.* Springer, 2014.

[17] M. Han and K. Daudjee, "Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing sys." *VLDB*, vol. 8, no. 9, 2015.

[18] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *VLDB*, vol. 5, no. 8, 2012.

[19] C. Mayer, M. A. Tariq, R. Mayer, and K. Rothermel, "Graph: Traffic-aware graph processing," *Trans. on Parallel and Distributed Sys.*, vol. 29, no. 6, 2018.

[20] A. Arasu, S. Babu, and J. Widom, "The cql continuous query language: Semantic foundations and query execution," *VLDB*, vol. 15, no. 2, 2006.

[21] E. Wu, Y. Diao, and S. Rizvi, "High-performance complex event processing over streams," in *Int. Conf. on Management of Data*, ser. SIGMOD '06. ACM, 2006.

[22] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. White, "Cayuga: A high-performance event processing engine," in *Int. Conf. on Management of Data*, ser. SIGMOD '07. ACM, 2007.

[23] Y. Mei and S. Madden, "Zstream: A cost-based query processor for adaptively detecting composite events," in *Int. Conf. on Management of Data*, ser. SIGMOD '09. ACM, 2009.

[24] A. Artikis, M. Sergot, and G. Paliouras, "An event calculus for event recognition," *Trans. on Knowledge and Data Engin.*, vol. 27, no. 4, 2015.

[25] G. Cugola and A. Margara, "Tesla: A formally defined event specification language," in *Int. Conf. on Distributed Event-Based Sys.*, ser. DEBS '10. ACM, 2010.

[26] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and et al., "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *VLDB*, vol. 8, no. 12, 2015.

[27] U. Srivastava and J. Widom, "Flexible time management in data stream systems," in *Symp. on Principles of Database Sys.*, ser. PODS '04. ACM, 2004.

[28] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis, "Mizan: A system for dynamic load balancing in large-scale graph processing," in *Europ. Conf. on Computer Sys.*, ser. EuroSys '13. ACM, 2013.

[29] L. M. Vaquero, F. Cuadrado, D. Logothetis, and C. Martella, "Adaptive partitioning for large-scale dynamic graphs," in *Int. Conf. on Distributed Computing Sys.*, ser. ICDCS '14. IEEE, 2014.

[30] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *Operating Sys. Design and Impl.*, ser. OSDI'14. USENIX, 2014.

[31] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking distributed stream data processing systems," in *Int. Conf. on Data Engin.*, ser. ICDE '18. IEEE, 2018.

[32] F. McSherry, M. Isard, and D. G. Murray, "Scalability! but at what cost?" in *Hot Topics in Operating Sys.*, ser. HOTOS '15. USENIX, 2015.

[33] C. Aggarwal and K. Subbian, "Evolutionary network analysis: A survey," *ACM Comput. Surveys*, vol. 47, no. 1, 2014.

[34] M. Gupta, C. C. Aggarwal, J. Han, and Y. Sun, "Evolutionary clustering and analysis of bibliographic networks," in *Int. Conf. on Advances in Social Networks Analysis and Mining*, ser. ASONAM '11. IEEE, 2011.

[35] M. Gupta, C. C. Aggarwal, and J. Han, "Finding top-k shortest path distance changes in an evolutionary network," in *Int. Conf. on Advances in Spatial and Temporal Databases*, ser. SSTD '11. Springer, 2011, pp. 130–148.

[36] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen, "Chronos: A graph engine for temporal graph analysis," in *Europ. Conf. on Computer Sys.*, ser. EuroSys '14. ACM, 2014.

[37] D. Sengupta and S. L. Song, "Evograph: On-the-fly efficient mining of evolving graphs on gpu," in *High Performance Computing*, ser. ISC '17. Springer, 2017.

[38] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen, "Kineograph: Taking the pulse of a fast-changing and connected world," in *Europ. Conf. on Computer Sys.*, ser. EuroSys '12. ACM, 2012.

[39] C. Song, T. Ge, C. Chen, and J. Wang, "Event pattern matching over graph streams," *VLDB*, vol. 8, no. 4, 2014.

[40] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *VLDB*, vol. 5, no. 9, 2012.

[41] J. Pérez, M. Arenas, and C. Gutierrez, "Semantics and complexity of sparql," *Trans. on Database Sys.*, vol. 34, no. 3, 2009.

[42] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus, "C-sparql: Sparql for continuous querying," in *Int. Conf. on World Wide Web*, ser. WWW '09. ACM, 2009.

[43] E. Della Valle, S. Schlobach, M. Krötzsch, A. Bozzon, S. Ceri, and I. Horrocks, "Order matters! harnessing a world of orderings for reasoning over massive data," *Semantic Web*, vol. 4, no. 2, 2013.

[44] A. Margara, J. Urbani, F. van Harmelen, and H. Bal, "Streaming the web," *Web Semantics*, vol. 25, no. C, 2014.

[45] H. Beck, M. Dao-Tran, T. Eiter, and M. Fink, "Lars: A logic-based framework for analyzing reasoning over streams," in *Conf. on Artificial Intelligence*, ser. AAAI'15. AAAI Press, 2015.

[46] A. Margara, G. Cugola, D. Collavini, and D. Dell'Aglio, "Efficient temporal reasoning on streams of events with dotr," in *Extended Semantic Web Conf.*, ser. ESWC '18. Springer, 2018.

[47] R. Angles and C. Gutierrez, "Survey of graph database models," *ACM Comput. Surveys*, vol. 40, no. 1, 2008.

[48] N. P. Schultz-Moller, M. Migliavacca, and P. Pietzuch, "Distributed complex event processing with query rewriting," in *Int. Conf. on Distributed Event-Based Sys.*, ser. DEBS '09. ACM, 2009.