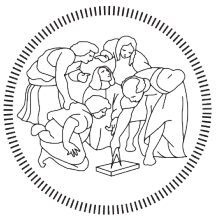

POLITECNICO
MILANO 1863

Scuola di Ingegneria Industriale e dell'Informazione
Dipartimento di Elettronica, Informazione e Bioingegneria
Corso di Laurea Magistrale in Computer Science and Engineering

Master Degree's Thesis

Design and Implementation of FlowGraph, a Distributed Framework for Temporal Pattern Recognition in Graph Data Structures

Supervisor
Prof. Alessandro Margara

Co-supervisor
Prof. Matteo Rossi

Candidate
Pietro Daverio
899800

Academic Year 2019–2020

Abstract

Graph data structures model relations between entities in many diverse application 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, virtually all applications are dynamic in nature, and this reflects to graphs that continuously evolve over time. 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 distributed framework for temporal pattern recognition in large scale graphs. We thoroughly evaluate the performance and scalability of FlowGraph with various workloads. 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.

Contents

List of Figures

List of Tables

Chapter 1

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.

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 [37] enable scalable distributed graph computations through a programming paradigm known as *think like a vertex* (TLAV) [52], introduced in 2010 with the Pregel system [46]. 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 [20, 24]. Modern big data processing platforms such as Apache Spark Streaming [85] and Apache

Flink [16] 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, which are then 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 [55, 35, 22], 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 [8, 27]. 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 distributed processing framework to detect 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. It exploits temporal properties within patterns to defer the execution of expensive computations, to sustain a high rate of changes.

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 patterns to further reduce processing time when possible.

The remainder of this paper is organized as follows. Chapter 2 presents background information and motivates our work. Chapter 3 introduces our data and processing model, and Chapter 4 provides their formal semantics. Chapter 5 illustrates the design and implementation of FlowGraph, and Chapter 6 evaluates its performance. Chapter 7 surveys work that is related to our proposal. Finally, Chapter 8 provides some conclusive remarks, suggesting possible future research directions.

Chapter 2

State of the Art

Our work is at the intersection of two research fields: graph processing and stream processing (in particular, pattern recognition over streams of events). Objective of this chapter is to provide a brief overview on the current state of research on Data-Intensive models and to show why a new model is advisable to analyze graphs evolution.

2.1 Graph Processing

Graph processing becomes challenging as graphs grow in scale. The processing of such large scale graphs cannot be handled with traditional mainstream parallel computation algorithms [45]. For example, social networks with billions of users and interactions could not fit in a single machine memory. A common approach is to partition the graph on a cluster of machines and process it. In other applications, graph are an intermediate representation of an extensive distributed pipeline, employed due to their characteristic of model entities and their relations. Also in this case data is partitioned in order to limiting data transfer, thus time cost.

Therefore, writing distributed graph application is inherently hard because it requires to deal with execution parallelization, data partitioning and communication management among cluster machines.

These challenges has found the interest of both industry and academic community that have produced a notable set of researches and papers on distributed graph processing in the last years.

In this section we present several popular high-level programming abstraction [37] and architectural choices for graph processing.

2.1.1 Vertex-Centric abstractions

Vertex-centric model is also known as *think like a vertex* (TLAV) or *vertex-oriented*, it is the established approach to design scalable distributed graph computations [52], and was introduced in 2010 with the Pregel system [46].

Many variants of TLAV graph processing model exist. For instance, some systems introduce multiple phases within each superstep, as in *Scatter-Gather* approach [60] and the gather-apply-scatter (GAS) model [28].

Vertex-centric

TLAV forces the user to express the computation from the point of view of vertex. It requires a static graph in which unique-ID vertices has local state values and the list of out-going edges with local values. The model is defined on oriented graphs, but it can be extended to undirected ones.

Pregel computation is *Bulk-Synchronous-Parallel* (BSP) [26, 75], therefore the entire computation can be split into phases, synchronized by synchronization barriers, also called *superstep*. BSP relies on a master-slave architecture for synchronization.

User defines a vertex-centric function, also called *kernel* that takes as input the state of current node, superstep number, the list of outgoing edges and the list of incoming messages to the node. Kernel can modify vertex values and it yields the set of messages directed to outgoing edges.

Vertices can be *active* or *inactive*. Inside each superstep, kernel function is executed in parallel on each active vertex. Initially, all vertices are active. Vertices can vote to halt computation becoming inactive. Inactive vertices becomes active if they receive any message. Vertex-centric function is applied only to active ones. Overall computation terminates when all vertices, for all superstep execution, does not produce any message and all nodes has become inactive. The final result of computation is the set of vertex values at the end of last iteration. We provide an algorithmic view of the described vertex-centric semantic model:

Algorithm 1: Vertex-Centric model

```
1  input: G(V, E)
2  vertices_active ← V
3  step ← 0
4  while vertices_active != Empty do
5  |   outbox ← Empty
6  |   foreach v ∈ vertices_active do
7  |   |   inbox_v ← getMessages(inbox, v)
8  |   |   if inbox_v.hasMessages() || step == 0 then
9  |   |   |   outbox+ : compute(v, inbox_v)
10 |   |   end
11 |   end
12 |   inbox ← setInbox(outbox)
13 |   step ← step + 1
14 end
```

To exemplify vertex-centric semantic model presenting *Google PageRank* in Figure 2.1. PageRank is based on the assumption that more important websites are likely to receive more links from other websites. Thus, it estimates the importance of a website by counting the number and quality of links (edges) to a page (vertex). Calling In_v the set of vertices with an edge pointing to v and $D(i)$ the outdegree of i, PageRank is defined on vertex v as

$$PageRank(v) = \frac{0.15}{n_{vertices}} + 0.85 * \sum_{i \in In_v} \frac{PageRank(i)}{D(i)}$$

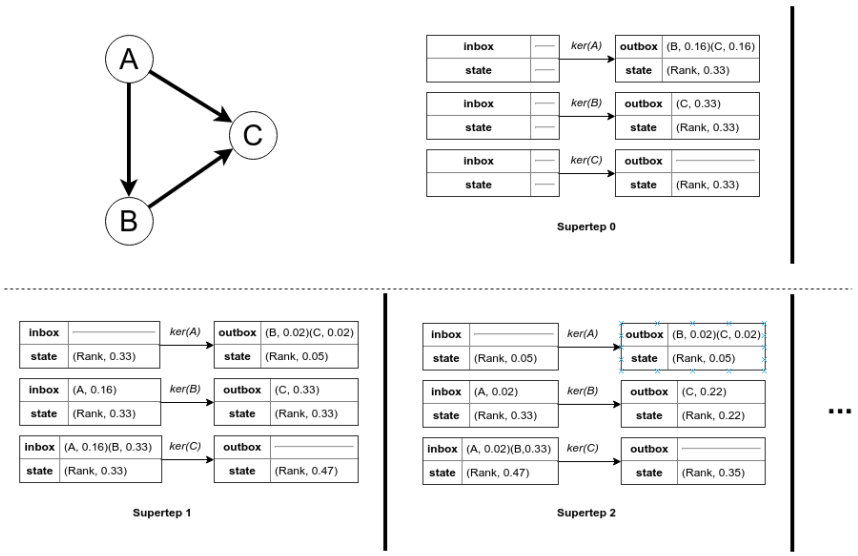

Figure 2.1: Example of first 2 iterations of PageRank [46] execution on a 3 vertices graph.

Each vertex associated PageRank value is initially set to $1/n_{vertices}$ and $PageRank/D(i)$ is sent on outgoing edges (Figure 2.1, Superstep 0). From Superstep 1, PageRank value is updated according to definition, aggregating values from incoming edges and spreading updated $PageRank/D(i)$ on outgoing edges. Vertex rank converges after several iterations [46].

Scatter-gather

With Scatter-Gather approach user has to implement two higher-order functions (*scatter* and *gather*). Both are executed on each active node inside a superstep.

Scatter function, executed in scatter phase, define the messages that will be sent along out-going edges. Scatter function takes as input vertex and return the list of messages to be sent. Gather function, executed in gather phase, takes as input vertex state as well as input messages returning an updated state of vertex values.

As in vertex-centric model, the two higher order functions are executed on the set of active nodes. All graph vertices are active in the first iteration, while only vertices with incoming messages are active from the second iteration (non-active nodes are inactive). Computation halts when all vertices has become inactive, so when no messages are sent inside the superstep.

Thus, the main difference of Scatter-Gather model with respect to vertex-centric one is the separation of receiving (gather) and sending (scatter) phases of messages on single vertex inside the superstep, not allowing sending messages during gather function execution and forbidding to access incoming messages in scatter phase.

Separating the two phases can make some programs easier to read, but it can also positively impact memory requirements, since it doesn't need concurrent

Algorithm 2: Scatter-Gather model

```
1  input: G(V, E)
2  vertices_active ← V
3  step ← 0
4  while vertices_active != Empty do
5  |   outbox ← Empty
6  |   foreach v ∈ vertices_active do
7  |   |   if inbox_v.hasMessages() || step == 0 then
8  |   |   |   outbox+ : scatter(v, inbox_v)
9  |   |   end
10 |   |   v.state+ : gather(inbox_v, v)
11 |   end
12 |   step ← step + 1
13 end
```

access both on *inbox* and *outbox* data structures. On the other hand, separating the two phases, all algorithms that send messages based on the content of received ones, will requires to store a partial result from gather phase locally to the vertex in order to retrieve it later in scatter phase, making algorithm harder to understand and forcing more I/O operations to store and retrieve values [60].

GAS

Finally, GAS model, introduced for the first time in Powergraph [28], distinguish between four phases: *Gather*, *sum*, *Apply* and *Scatter*.

In this case, gather function is not defined at vertex level but at incoming-edge level, in other words, inside the same superstep, it is executed for each incoming edge. Results of edges computations are collected in the next phase by sum function, that it is required to be associative and commutative. The result of sum and the state of vertex are passed to apply function that compute new vertex values. At the end of sum phase, scatter phase execute a user-defined function that sends new messages along outgoing edges, based on updated vertex state.

Applying gather phase on edges, as happen in GAS model, it is possible to parallelize computation on edges mitigating the problem of degree *skew*, typical of the majority of real graphs (high-degree vertices usually takes more execution time creating a computational imbalance behaving as *stragglers*).

If an algorithm cannot be decomposed in GAS steps with an associative and commutative sum, implementation can just behave as a mere emulation of a vertex-centric computation loosing benefits on skew mitigation but continuing to pay higher overheads in memory and communication, due to the handling of more algorithm phases. Therefore, some studies propose a *differentiated vertex computation* model in which vertex-centric is applied to low-degree vertices while GAS is preferred with high-degree ones [17].

Algorithm 3: GAS model

```
1  input: G(V, E)
2  vertices_active ← inputVertices
3  step ← 0
4  while vertices_active != Empty do
5      foreach v ∈ vertices_active do
6          foreach e ∈ v.incomingEdge do
7              aggregate_v ← sum(aggregate_v, gather(v, inbox_v, e))
8          end
9          v.state+ : apply(aggregate, v)
10         outbox+ : scatter(aggregate, v)
11     end
12     inbox ← setInbox(outbox)
13     step ← step + 1
14 end
```

Subgrap-centric

Other models provide subgraph-centric primitives that enable asynchronous evolution of subgraphs to some degree [70, 33], or mimic shared memory programming abstractions to ease the development of algorithms [44].

Subgraph-centric models extends the concept of vertex-centric model to subgraphs, considering subgraph the target of user-defined function. In other worlds, kernel function is applied to the entire subgraph instead of single vertex, without changing the semantic.

The most common subgraph-centric model is *partition-centric*. With this approach, subgraph taken as target of kernel correspond to graph partition. The main advantage of this approach is the reduction of communication overhead allowing kernel to access all vertices in the same partition at once avoiding message passing on local memory scope. All vertices that are not part of current partition but that are referenced by outgoing edges from partition's vertices are called *boundary vertices*. Boundary vertices provide a local copy of vertex value that will be kept updated from the partition in which vertex reside.

Subgraph-centric models are able to save communication time if partitioning are formed in order to minimize edge cuts, however it looses its performance benefits with respect to vertex-centric approach if the graph is poorly partitioned.

Furthermore, for most of users and graph computations, TLAV approach is more intuitive rather than *Think Like A Partition* model.

2.1.2 Other programming abstractions

Research community provides some less known or adopted alternatives to vertex-centric or subgraph-centric approaches. We will review briefly some of them for completeness.

Graph Traversals

Implemented by *The Gremling* [63] graph traversal machine in *Apache Tinkerpop*, it provides distributed traversals with a Bulk Synchronous Parallel computational model.

Traversers are modeled as messages and walks throughout a graph one time for each superstep. Vertices receives traversers, execute a user defined function and optionally generates new traversers. Halted traversers are stored in vertex attributes and the entire computation halts when no traversers are generated in the last superstep.

Filter-Process

Proposed in *Arabesque* system [74], it basically consists into two functions: *filter* and *process*.

Filter function detect and select, from the input graph, the subgraph (also called *embedding*) that become the target of process function. Process execute a function on the embedding and optionally produce an output.

Computation is performed in several steps, following a BSP execution model. In the first *exploration step* the candidate set contains all vertices (or edges) of the input graph. From the second superstep, input-set is formed from selected set of the previous superstep and the computations halts when no more embedding have to be extended.

This model is well-suited for graph pattern mining problems as spam detection and semantic data processing that are challenging to be express with a vertex-centric model.

2.1.3 Timing

In graph processing, execution models, as presented in 2.1.1, are uncouple from timing and scheduling logic.

Timing models define the order for active vertices execution. There are three main models [52]: *synchronous*, *asynchronous*, *hybrid*

Synchronous

Based on *Bulk-Synchronous-Parallel* (BSP) [26, 75], it enables active vertices to be executed in parallel within a superstep, thus no assumption on execution order can be made inside the superstep. Master handles the global synchronization barrier that guarantee no worker can begin a new superstep before the end of the previous one.

Synchronous systems are conceptually simpler imposing determinism in the number of steps, message exchanges and results. They also demonstrate to be scalable with respect to number of vertices [46].

Synchronization imposes some performance drawbacks. For instance, on shortest-path algorithm, it has been observed that 80% of total running time has been taken by synchronization overhead on a partitioned graph execution [71]. In general, iterative algorithms particularly suffer the *straggler* problem (section 13).

Asynchronous

With asynchronous model, synchronization barriers are not present, allowing scheduler to dynamically generate and change execution schedules. This solution solve the *straggler* problem increasing resource utilization, especially with I/O bound and imbalance algorithm [81].

This model is more complex to manage and optimize, algorithms are less intuitive to write and follow, also due to an usual non-determinism in execution. Also results could converge to different values for different executions. Usually, asynchronous execution is implemented in shared-memory systems models and it also requires to manage data-race conditions [79].

Hybrid

Hybrid systems try to take advantage of both timing models. Rather than a model, hybrid could be considered as a set of solutions and strategies that may be rather different from each others.

One strategy to reduce the cost of global synchronization imposed by *barriers* are *pseudo-supersteps* [61], that allows to decompose the global superstep into several intermediate steps, usually local to partition.

Other strategies rely on the idea of taking advantage of asynchronous execution and dynamic scheduling, inside the single graph partition between two synchronization barriers. This is a design choice for *partition-centric* model (see section 14). In this case, barrier execute a synchronous step on boundary nodes and requires user definition of two kernel functions, one for synchronous superstep, the other for asynchronous *local* execution [79].

Finally, some systems overcome straggler problem converging asynchronously after several synchronous iterations [79].

It has been proved that hybrid models can reach performance comparable to asynchronous ones, but they remains more complicated to study and observe with respect to synchronous model [79].

2.1.4 Partitioning and Communication

A common problem in graph processing is how to split a large-scale graph into parts, to be placed in a distributed memory. In other words, how to partition a graph. As already mentioned (see section 14), good partitioning could leads to better performance [65, 76]. Good partitioning should evenly distribute workload among workers minimizing communication efforts, thus reducing edges or vertices cuts and message exchanged between partitions maximizing data locality. The *K-way graph partitioning problem* is NP-complete [4] but there are some strategies and heuristics that allows to improve partitions quality such as METIS algorithms suite [39].

METIS may allow to obtain near-optimal partitioning in TLAV frameworks but it usually requires long pre-processing phase to calculate and reallocate vertices on partitions. Consequently, this solution may not fit the majority of applications, especially on large scale, highly-connected graphs. Some distributed heuristics has been built on this METIS that proved to speed up particular classes of graph computations [62].

Some frameworks support dynamic re-partitioning strategies for vertices in order to mitigate computational imbalance. Dynamic re-partitioning poses different issues, from the selection of vertices to move, on the implementation of a logic that define in which moment and toward which destination move vertices and the guarantee of reachability before and after the relocation by all system workers [65]. This set of techniques proved to reduce network I/O but introducing a computational overhead that usually overcome benefits [57].

Another common approach, to decrease data communication between workers is graph partitioning by edges instead of vertices, creating duplicates of *border vertices*. Values of duplicated vertices are kept consistent by framework [29].

Another problem, bounded with partitioning, is communication, in other terms how data is shared between vertex programs.

The most common communication model in TLAV framework is *message passing*. A message is composed by message data and destination address, moreover destination is obtained from the set of vertex outgoing edges. At the end of iteration, generated messages are dispatched by worker toward destination inboxes, that can be on the same or different partitions [46].

An improvement to this solution is *message-batching*, in which generated messages are collected in a buffer and sent as soon as its size reaches a threshold, otherwise at the end of superstep. This solution not only improve performance increasing network efficiency, but it also allows to decrease memory requirements of synchronous computations [66].

Some frameworks also implements a *combiner* in order to de-duplicate repeated messages sent to different destinations reducing network usage. Combiner operations can be decomposed into two steps: (i) *sender-side aggregation* combines repeated messages value into one and (ii) *receiver-side scattering* scatters message on receiver side, forwarding a copy of the message for each destination.

Single machines frameworks usually benefit most from a *shared-memory* model, in which vertex program kernel can directly access and modify all vertices in the scope [69]. This solution have to guarantee consistency and avoidance of data-race conditions that are instead intrinsic in synchronous message passing model.

2.2 Stream Processing

Traditional DBMS, also known as *Human-Active Database-Passive* (HADP) systems, become unfit for a rising number of modern applications that require to analyze large amount of dynamic data generated as a continuous dataflow from different distributed sources. In particular, traditional DBMS are limited by the need of storage and indexing of incoming data before running any execution on it and the need to trigger the execution from an external, asynchronous (human) entity. Many evolutions as well as different models has been proposed in the last decades [20] and we will review briefly them pointing out their application fields and limits.

A first step forward came from active databases [51]. They allow to automatically execute a set of actions on data as a given condition is matched. This behavior, also known as *reactive* is based on the general *event-condition-action Knowledge Model* [58] that defines which kind of *event* to take into account

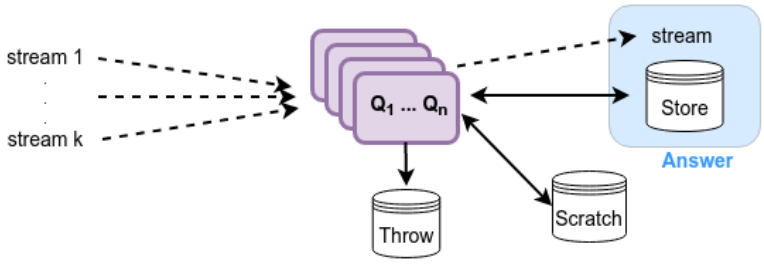

Figure 2.2: Babu and Widom proposed architecture for Continuous Stream Processing

and which *condition* the selected event have to satisfy in order to trigger the execution of a user defined SQL *action*. This solution allows to automate data processing but is unable to overcome the limitation posed by a single logical persistent storage with respect to the possibility to handle frequent events.

In this context, *Data Stream Management Systems* (DSMSs) [9] offers a different model, based on continuous queries, that, once deployed, can produce results processing continuous data stream.

The general architecture for DSMS model, as defined by Babu and Widom [10] is provided in Figure 2.2. This model requires as assumption the definition of queries over data streams only but it can be extended to include conventional relational symbols. Any time an incoming tuple t is notified to query Q, the latter can take several actions based on t: it can update answer A appending new tuples to *stream* of responses or updating *Store* (data-structure designed to store information that belongs only temporary to answer A), it also may put t, or data derived from t, into *Scratch*. Scratch collects data that isn't part of solution but that can be retrieved later. Finally we can discharge useless data sending it to Throw. Therefore, query Q consumes input token and it can access and modify data from Store and Scratch memory.

An improved triggering model for active databases has been proposed by Babu and Widom, based on *Alert* approach. Continuous queries are allowed on event streams together with conventional tables. Stream and store, composing the Answer, may remain empty if trigger's conditions aren't fulfilled, otherwise the generated SQL action itself is appended to answer stream. Employing this approach for triggering, it is possible to express complex multi-table events and conditions as well as benefit from efficient data management and processing techniques that have been developed for continuous queries [10].

Closely linked to DSMS concept of continuous queries, from the perspective of programming language paradigm, dataflow programming abstraction largely found adoptions in Batch and Stream Computing such as Internet of Things, mobile, clickstream, sensors and market analytic fields [40].

Dataflow Architecture Dataflow programming paradigm, also called *datastream*, provides a programming architecture abstraction to model programs as directed graphs in which data flows between operators.

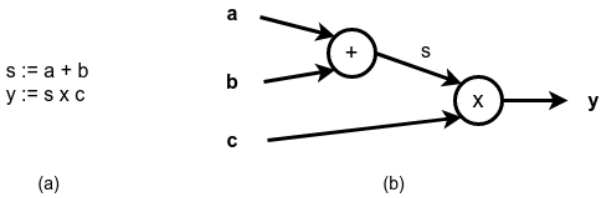

Figure 2.3: Dataflow example of (a) parallel instructions written in pseudocode and (b) the corresponding dataflow graph.

Arcs represent the data dependencies between operators and behave as an unbounded *First-In First-Out* (FIFO) queue while each node is an operator that may be a single instructions [41] or a section of sequential instructions [42]. An example of dataflow is provided in Figure 2.3. Arcs entering a node are called *input* arcs, while arcs exiting the node are *output* ones. *Firing set* for an operator is the set of input arcs that have to provide data to feed operator. When firing set has data ready to be process for all its arcs, operator become *firable* and system can schedule its execution. Execution consumes data from firing set, process it according to operator logic and eventually produce new data on one or more output arcs. Finally, operator terminate and wait to become firable again [36].

A programming language supporting dataflow architecture should be able to expose graph and data dependencies. Despite data dependencies may be expressed in all classes of languages, they tend to infer different degrees of parallelism [36]. According to Wail and Abranson [78], a dataflow programming language should be (i) free from side effects, (ii) effects should be local, (iii) data dependencies should impose the scheduling, (iv) variables cannot be reassigned, (v) an ad-hoc notation for iterations is needed and (vi) operators shouldn't be sensible to procedure's history. Given these principles, functional programming has gain popularity in this context due their characteristics of avoiding side effects and reassignments, while imperative paradigm is often used to express loops [78].

2.2.1 MapReduce

Stream processing approach has been widely employed to develop many cluster computing frameworks such as *MapReduce* [61]. These systems let the user to write parallel computations extending a set of *higher-order* operators. Higher-order operators have to be customized to implement operator logic. In MapReduce data tokens are in form of key-value pairs and there are a fixed set of higher-order operators to extends, some of them are: (i) *input reader* that takes as input key-value token, (ii) *map* operator that takes as input a series of tokens, eventually returning a new token for each of them, (iii) *partition* operator to control data distribution, (iv) *reduce* operator that eventually produce a single token analyzing the entire stream and (v) *output writer* that define system sinks. MapReduce programming model with higher-order functional interfaces

has been widely adopted in modern big-data frameworks [20].

It can be demostrated that MapReduce is able to emulate arbitrary distributed computation. In fact, a generic distribute computation can be seen as the composition of two classes of operation: (i) one is the computation local to a computing node, that can be modeled by a *map*; (ii) the other is occasionally exchange of messages between nodes, that can be modeled with a *reduce*. Thus, the distributed computation could be modeled with a BSP [75] model in which each *reduce*, behaving as synchronization barrier, collect the entire computation state and messages that nodes have to exchange, redistributing them in following superstep. However, writing-out the entire state after each step could be inefficient resulting in high latency and network bottleneck [86].

While MapReduce framework has proven to provide a parallel and scalable solution for a large set of batch processing, scaling out computation to multiple nodes, it is not sufficient by itself to cover the diversity of computing workload that characterize modern applications as iterative graph processing or interactive SQL-like queries.

2.2.2 Spark

As mentioned in 2.2.1, MapReduce model could potentially emulate any distributed computation, but it may be inefficient for many types of workloads. On the other hand, most of big-data applications need to handle different workloads in the same application. One common solution is to pipeline different specialized frameworks, each one addressing one specific computation workload. For example, MapReduce can be used to load and preprocess a dataset that will be fed as input to an interactive SQL queries system, on top of which some specialized machine learning framework can perform iterative ML computations. This kind of approach may result in many complexities and inefficiencies. Firstly, users need to interface different systems and run different distributed frameworks. Secondly, I/O operations require the storage and retrieval of intermediate results increasing in latency and network usage [86].

Spark tries to overcome the above mentioned limitation providing a MapReduce model over *Resilient Distributed Dataset (RDD)* [83] abstraction. RDDs are immutable, fault tolerant collections of data, partitioned across a cluster that can allow parallel data processing.

User controls *driver* program in which he defines operations on RDDs. Spark API provides *Spark Context* to interface driver with the *Spark Cluster* (Figure 2.4). Spark program, also called *Job*, is submitted to Spark *Cluster Manager* that controls cluster resources. Job is converted into its corresponding Directed Acyclic Graph (DAG) to determine RDD dependencies and optimize execution plan. The resulting execution plan is composed of a set of *tasks*. *Spark Scheduler* schedules tasks on cluster *executors* coordinating the distributed execution. Finally, computation results are sent back to user program.

Any change in RDD data as well as any transformation applied to an RDD will result in the definition and re-build of a new RDD [83]. However, RDD permits to leverage distributed memory, shifting the computation as local to data as possible, also allowing local storage and the use of intermediate results through caching mechanisms. Data reuse is common in many iterative machine learning and graph algorithms. For example, *PageRank* stores intermediate computed

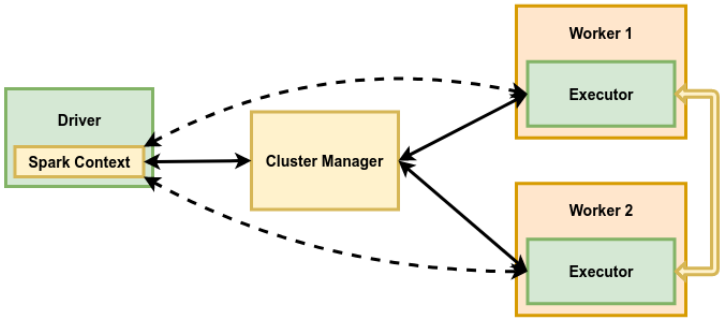

Figure 2.4: Spark cluster composed of two workers machines serving one driver.

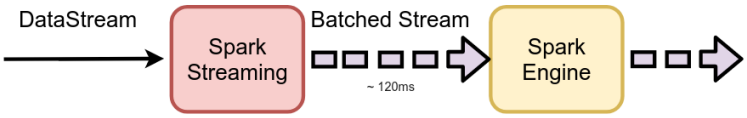

Figure 2.5: SparkStreaming library collects continuous inputs streams in data batches every 120ms before processing

Rank to retrieve it in the following iteration. Data sharing, among different iterations, provides large speedups also in interactive queries algorithms [82].

Users can operate on RDD using Spark functional programming API to define functions that will be passed to Spark Cluster. The chain of higher-order functions will be optimized and the corresponding execution DAG dynamically scheduled. Finally, the result is returned to client. Spark lazily evaluates RDDs finding an efficient dataflow plan through operation reordering and grouping [85].

It is possible, with the use of RDDs, to build a variety of higher level libraries targeting many of specialized computing engines use cases. Spark enables to potentially express any distributed computation since it provides a complete MapReduce model. It has been demonstrated that RDDs, giving applications control over common bottle-neck resources in clusters—network and storage I/O, make it possible to mimic most of optimizations that characterize specialized systems reaching comparable performances [83].

Spark provides a set of libraries to operate with different classes of dataflow, each one presenting a higher abstraction of data-structures and operations based on underling RDDs dataflow (Figure 2.6 (A)).

- SparkSQL [6] provides a relational queries pararadigm with cost-based optimization of queries. Data is organized in *dataframes* that are RDDs of records structured in tabular layout.

- Spark Streaming [85], exemplified in Figure 2.5, is based on the concept of *discretized streams*. Data stream is split in small chunks on time basis.

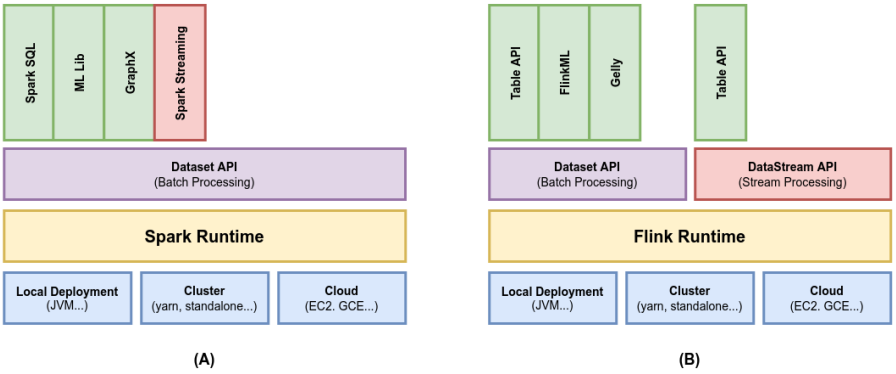

Figure 2.6: Framework component stack comparison between Spark (A) and Flink (B). Spark provides a compatibility library to handle DataStreams while Flink naively support DataStream computation.

Standard collecting window is 200 ms. As time wall is reached, batch is sent into the system and, combined with RDDs state, it produces results.

- GraphX [29] is the library dedicated to graph processing. General-purpose join and aggregation strategies do not leverage the common patterns and structure in iterative graph algorithms and therefore they may miss important optimization opportunities. Thus, despite graph dataset is stored on top of RDDs, GraphX implements many graph processing optimizations and takes advantage of peculiar data locality of graphs. GraphX implements Pregel model and optimizations such as GAS approach, advanced partitioning strategies and messages aggregations. It also proved to scale and perform on par with many specialized graph processing solutions.

- MLlib [86] provides the implementation of several machine learning algorithms for distributed model training.

Despite implementation of data stream processing applications is possible, it may suffer from limitations from high latency and performance constraints, due to batching strategies and immutability of RDD. Moreover, Spark lacks of notion of time and windowing techniques that are typical of some stream applications. These characteristics, in spite of allowing some batching optimizations, make Spark unfit to fulfil some application domains as real time continuous streams of data. In general, RDDs are less suitable in handling asynchronous, fine-grained updates to shared state [83].

2.2.3 Flink

While batch processing has as target to optimize throughput and scalability of high volumes of static data, stream oriented models focus on lowering latency for application in which response time is an important factor. Trade-off between the two classes of models has been studied and it has been proposed an hybrid architectural pattern called *lambda architecture* [49]. The idea is to convey datastream through two different paths for computation: a fast one for a timely

approximated result and the other for a batch offline computation to get late, but accurate, results. Still, this solution, implemented in several batching systems, suffers from high latencies imposed by stream batching and it adds complexity by the need of handling two parallel systems.

Apache Flink [16] is an open-source system for processing streams and batch data. It allows to deploy any application that can be expressed as a pipelined fault-tolerant dataflow. Therefore, as Spark, it includes processing of historic data in batches and executing of iterative algorithms typical of graph-processing and machine learning application. Furthermore, with respect to Spark, it also supports applications of real-time analytic and continuous data pipelines in a time awareness fashion that addresses the continuous timely nature of produced data.

Flink processing and execution models make no distinction between real-time processing of latest events and historical data, offering a unified data stream processing model for both continuous streams and batches. Flink also offers a flexible windowing mechanism for events based on time that allows to perform both early approximate as well as delayed and accurate results.

A runtime program can be modeled as an Acyclic Directed Graph (DAG) in which stateful operators are connected by streams. Flink cluster is essentially a master-slave architecture in which a client application receives user defined program code, transforms it into a corresponding dataflow and submits it to JobManager (master). The latter schedules and coordinates the distributed execution controlling TaskManagers (slaves) that execute one or more operators and use network connections to exchange data streams between operators.

Batch processing is modeled as a stream processing on bounded streams where ordering and time of records does not matter. Thus, records are theoretically considered all in the same time window. Therefore, batch processing has an extended API that allows operations like joining and grouping and implements dedicated scheduling strategies. For this reason, users can decide between two different core API (Figure 2.6 (b)): a (i) Datastream API for a timely process of potentially unbounded datastreams with time windowing and stateful operators; and a (ii) Dataset API for batch computation. On top of them, Flink provides higher-order libraries for machine learning, SQL-like queries, graph processing (called Gelly) and Complex Event Recognition support.

In datastreams, time can be defined as *event time* (or source time) that represents the generation time of an event or *processing time* (and ingestion time) that represent the wall-clock time of the machine that is starting processing (or is receiving) the incoming event. Event and processing time may differ arbitrary [3], moreover, considering event time, incoming events may arrive out-of-order, thus Flink implements a *watermark* strategy to keep track of global system progresses and ditch too delayed events. On the other hand, opting for processing time guarantees lower latency since incoming streams are considered in-order and there is no need to wait for possible out-of-order events.

2.2.4 Iterative dataflow

The problem of efficient iteration in data-parallel processing frameworks has been addressed by different strategies. Common MapReduce frameworks, as Hadoop [13], do not provide explicit support for iterations and they suffer from performance degradation in case of iterative steps emulations [15] since a new

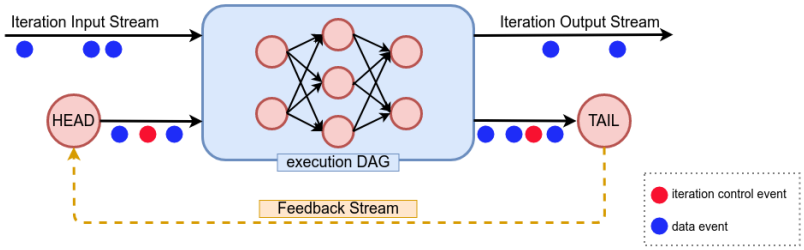

Figure 2.7: Example of Flink feedback-edge stream in which data produced in current iteration by execution DAG is back-forwarded to feed the next step.

iteration will result in the submission of an entire new job and data reshuffle across partitions.

Haloop extends the Hadoop framework to offer an efficient loop-aware implementation by handling loop control and offering a programming interface to express iterations. It optimizes data reuse across iterations and minimizes reshuffling, caching and indexing of local data [13].

Spark allows to express loop through common iterative constructs like *for* and *while*. This iterative definition will result in the addition of operators to the running DAG. Spark's loops require to write intermediate computation state on new RDD [84]. RDDs have caching optimizations to store and retrieve intermediate results between iterations minimizing data reshuffling.

Flink uses *feedback edges* [56] strategy in which iterations steps are special operators that can contain an execution DAG. This allows to maintain a DAG-based runtime and scheduler in which *tail* and *head* tasks interact explicitly connected by feedback edges, as shown in Figure 2.7. A feedback stream connects the output of previous iteration with the input of the following one. For batch iterations, Flink allows to implement structured iteration logic using iteration control events [16]. Flink also introduces the concept of *delta iterations*, a form of explicit incremental iteration, that address the problem of immutable state, exploiting the sparse computational dependencies inherent in many iterative algorithms as in graphs processing [25].

2.2.5 Graph processing on top of Stream Processing

As we have seen in 2.2.4, dataflow is a suitable model to express iterations and graph processing algorithms and we also have seen in 2.2.2 and 2.2.3 how modern dataflow framework offers efficient graph processing libraries [29] [16] that allow to express different kind of structured iteration logic, such as BSP [75] taking advantage of sparse computational dependencies and maximizing data locality and reuse.

Although these systems have shown good scaling properties in analysis of static graphs, they are not designed to analyse the evolution of graphs over time. For instance, GraphX's immutable nature of RDD requires to re-instantiate data-structures also on fine grain graph update. Gelly, on the other hand,

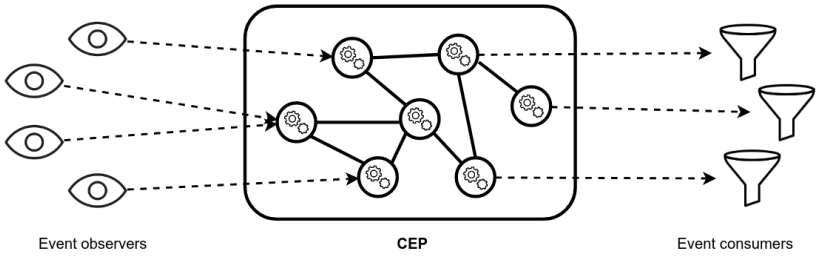

Figure 2.8: Complex Event Recognition (CER) representation showing distributed agents connected by an overlay network (solid arcs) and input and output system streams of events (dashed oriented arcs)

supports graphs updates but still does not provide a flexible timely model to study evolution for graph properties and topology.

2.2.6 Complex Event Recognition

Complex Event Recognition (CER) tries to offer a representation of datastream in which each data token has an associated semantic. Thus, incoming data tokens from external world, generated by *event observers*, are expressed as *notification* of events. CER system filters and combines events in order to identify and notify, as *complex events*, pattern of notifications interesting for the specific application. Finally, *sinks* are event consumers that collect those events that represent CER system output.

Inside, a CER system (Figure 2.8) could be seen as a network of *event processing agents*, connected through the *event processing network*. The system employs specialized routing and forwarding strategies to guarantee efficient processing of heterogeneous data sources and scalability.

CER processors was initially based on *message-oriented publish-subscribe* interaction model in which users define the information they are interest in and for which they want to be notified [64]. This model has been extended to allow the combination of events into composite ones.

Information can be filtered and combined expressing a *subscription* in terms of event topic or based on the content of subscription. In the former case, *topic-based* subscription model allows user to select the class of event they want filter choosing it from a predefined set of topics [64]. In the latter case, *content-based* filtering allows to select information also relying on the content of event notifications [2]. CER systems focus their attention on detection of event patterns based both on their content as well as ordering and sequencing relationship [19].

Different formalisms have been proposed for pattern specification and recognition [8, 27], ranging from regular expressions/timed automata [80, 14] to operator trees [54] and logic formulas [7, 19].

2.3 Motivations

As we have seen seen in Section 2.1, the established approach to design scalable distributed graph computations is known as *think like a vertex* (TLAV) [52]. Many variants of this model exist. For instance, some systems introduce multiple phases within each superstep, as in the gather-apply-scatter (GAS) model [28], others provide subgraph-centric primitives that enable asynchronous evolution of subgraphs to some degree [70, 33], or mimic shared memory programming abstractions to ease the development of algorithms [44].

In Section 2.2, we have analyzed stream processing systems and big-data processing platforms as MapReduce [61], Apache Flink [16] and Apache Spark Streaming [85]. 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 [20]. Another processing model, named Complex Event Recognition (CER) [24], looks for temporal patterns in the streams of input data.

Graph processing systems are designed to handle *static graphs*. 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. On the other hand, stream processing systems focus on streaming data and offer limited support for integrating static data stores, since they only manage the state that is needed to compute the desired results, such as window contents or partially recognized patterns.

2.3.1 Objective

In this work, we build on the CER model—recognition of temporal patterns—and augment it to support graph-shaped state and graph computations. We adopt a logic formalism that seamlessly integrates graph computations as predicates of the language. We present a prototype pattern recognition system that stores the graph state into the main memory of multiple machines, implements graph computations using TLAV algorithms, and distributes the state and processing associated with pattern recognition as much as possible.

Chapter 3

Data and Processing Model

Figure 3.1 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 [3] 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.[1]

FlowGraph provides a high-level language to define patterns. This section introduces the language features by examples while the following Chapter 4 formalizes their semantics.

3.1 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 a social media application, vertices can represent users

[1] Assumptions and mechanisms to cope with out-of-order arrivals of events have been discussed in the past and can be adopted to ensure this property [72].

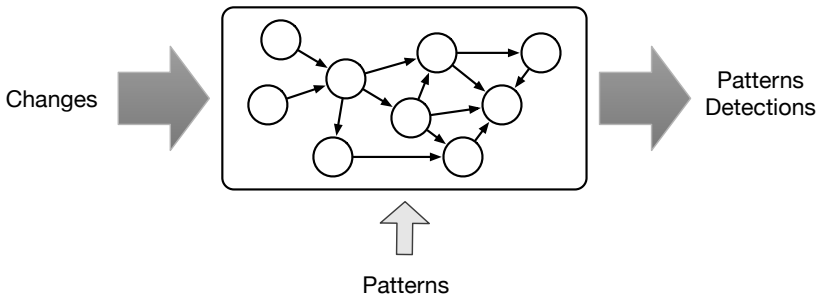

Figure 3.1: FlowGraph data and processing model overview.

and edges the relations among them. In this context, the labels associated with vertices can indicate properties of users, such as their name, nationality, and age, and the 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 algorithm 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 and 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.

3.2 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 predicate on the state of individual vertices or edges, or to aggregated values computed over the entire graph or parts of it. They can reference both the current state and the state at some previous point in time, and correlate their values.

In the remainder, we incrementally present the core language constructs that we offer to derive values from a labeled graph, and then we show how they can be combined to form individual clauses and complete patterns.

3.2.1 Computations

To derive new values from the ones explicitly defined for vertices and edges, FlowGraph supports vertex-centric computations, which promote parallel processing and proved to be efficient and scalable to large graphs. 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.

As an example, let us 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 already 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. For instance we provide an example of usage of max operator in pattern language:

```
g().compute(Max, $maxVal, [label=Height])
```

`g()` executes subsequent operations on the entire graph, `compute` invokes a vertex-centric computation identified by first compute's parameter while second parameter defines the variable associated with result label (variables are explained in section 3.2.6). In square brackets, the assignment list of computation-specific parameters.

3.2.2 Selection

FlowGraph provides selection primitives to isolate a subgraph based on the values of labels of vertices (`selectV` primitive) or edges (`selectE` primitive). Specifically, a `select` primitive takes in input a predicate—that is, a function that evaluates the state of vertices or edges—and returs 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. This is exemplified by the code snippet below. First, we `compute` the shortest path tree on the graph starting from vertex `v`, using `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`. Label `distance` represents the link between the shortest path computation and the subsequent selection: the computation assigns such label to each vertex in the graph, and the selection uses the label to isolate a subgraph. The algorithms in the standard FlowGraph library let the user customize the name of the labels

where they store their results, such that the results of multiple computations do not conflict.

```
g().compute(ShortestPath, $distance, [fromVertex='v235'])
   .selectV(distance < 10)
```

We may also want to select all vertices identified by at least one `active` edge, where `active` is a property associated with edges.

```
g().compute(ShortestPath, $distance, [fromVertex='v235'])
   .selectV(distance < 10)
   .selectE(active='true')
```

All the primitives that work on graphs can be applied to selected subgraphs. For instance, the following code snippet starts a community detection algorithm only on the selected subgraph.

```
g().compute(ShortestPath, $distance, [fromVertex='v235'])
   .selectV(distance < 10)
   .compute(CommunityDetection, $community,
            [numCommunities='5', type='k-clustering'])
```

3.2.3 Values extraction

Values 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: in this way, users always obtain the identity of the vertex or edge as part of

the extracted values.

To exemplify values extraction, given a graph in which each node represent a city, the following code snippet extracts all the name of cities and coordinates that have a path of less than 1000km from Milan. 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.

```
g().compute(ShortestPath, $distance, [fromVertex='id_milan'])
   .selectV(distance < 1000)
   .extractV(name, x_coord, y_coord)
```

3.2.4 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`, which computes exactly one output element for each input element according to a user-defined function, and `flatMap`, which computes zero,

one, or more output elements for each input element, 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 associated to `compute` result variable.

```
g().compute(ShortestPath, $distance, [fromVertex='v235'])

($distance)
   .lessThan($distance.value, '10')
   .avg($filtered.value)
```

3.2.5 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.

```
g().compute(CommunityDetection, $community,
            [numCommunities='5', type='k-clustering'])
   .subgraphByV($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: notice that this enables vertices to belong to more than one subgraph.

3.2.6 Variables

Using the `emit` or `compute` primitives, FlowGraph defines variables to bind values in different parts of a pattern. Variables can refer to graphs or values extracted from computations on graphs. FlowGraph interprets tokens inside single quotes as values ('value') and token preceded by $ as variable names ($variable), otherwise they are considered properties or labels names of graph. For instance, the following code snippet counts the number of people older than 20 from the largest community (or communities).

```
g().compute(CommunityDetection, $community,
            [numCommunities='5', type='k-clustering'])
   .subgraphByV($community)
   .emit($communityGraphs);
```

```
$communityGraphs
   .extractV()
   .count()
   .emit($communitySize);

$communitySize
   .max()
   .emit($maxSize);

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

The first part of the pattern performs community detection and groups vertices according the their value for the `$community` label variable. It associates such groups (graphs) with a variable `$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 refers to the three emitted variables. It starts from the graphs in `$communityGraphs` and selects the one (or ones) having maximum size. This selection makes use of the `$communitySize` and `$maxSize` variables previously emitted. Finally, the pattern selects and counts the vertices having a value greater than 20 for label `age`.

3.2.7 Temporal operators

FlowGraph lets users predicate on the temporal evolution of a graph by using variables to 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. FlowGraph support both natural time (seconds, hours, days) as well as other logical ordering of events.

For instance, the following snippet refers to the size of the graph (number of vertices) 10 seconds before the time of evaluation. The `ago` keyword allows to access old views of values associated to variable, yielding values valid for the time instant indicated by the offset. The offset is formed of a numeric value `t` and a time unit. Thus it returns the value of a label or computation as if it was performed `t` time units before the current time.

```
g().extractV()
   .count()
   .emit($graphSize);

($graphSize 10-s ago)
   .emit($previousSize);
```

Similarly, the following snippet computes the maximum size of the graph in the last 10 seconds. `Within` primitive returns the list of values that a variable assumed in a time window started `t` time units ago (included) and ending now (excluded), covering all values assumed by the variable in the time window.

```
g().extractV()
   .count()
   .emit($graphSize);

($graphSize within 10-s)
   .max()
   .emit($maxSize);
```

3.2.8 Pattern clauses

As explained at the beginning of this section, 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 code snippet defines a clause that is satisfied whenever (at least) one community larger than 20 is detected.

```
g().compute(CommunityDetection, $community,
            [numCommunities='5', type='k-clustering'])
   .subgraphByV($community)
   .extractV()
   .count()
   .evaluate("LargeCommunity", >, 20);
```

We will extend the language in the future to let users build richer notifications, for instance to output a custom result for each community in the example above. The problem of identifying the data that contributes to the triggering of a pattern and referring to it when building notifications has been widely studied in the context of complex event recognition under the name of *event selection* [27]. We plan to inherit established solutions from that domain.

3.2.9 Triggers and conditioned executions

We decided to expose some lower level language features in order to provide the possibility for user to define events and conditions patterns have to react to, triggering patterns or portion of them.

Currently, supported classes of events are vertex (and edge) *creation*, *modification*, *deletion* and *periodic-actions*. Periodic-actions can be seen as events that enter the system on timely base. It can be observed how creations and deletions of graph entities change the graph topology while we refer to any change in properties of labels as a modification event.

`Trigger` allows to avoid executions that are not useful for the definition of pattern results or are too heavy to be recomputed every time. In the following example, community detection is executed every 10 seconds, while an increase in number of vertices is notified promptly.

```
trigger(10-s)
   .g()
   .compute(CommunityDetection, $community,
            [numCommunities='5', type='k-clustering']);

trigger(vertex insert)
   .evaluate("GraphIncreased");
```

In most situations, it may be possible to automatically decide to filter out some pattern executions without changing the semantics of pattern. For instance, we know that some computations, as community detection algorithm, doesn't change their results without a topology changes. We are planning to integrate those automatic static optimization techniques in FlowGraph.

It is also possible to decide to evaluate a pattern based on the outcome of another pattern execution. To exemplify the concept of *conditioned execution*, we provide a pattern in which we perform a *PageRank* only if the number of total edges in the graph has increased with respect to 10 seconds before.

```
.g().compute(OutgoingEdges, $outDegree)
    .emit($graph);

.collect($outDegree).reduce(count)
   .emit($totalEdges);

.collect($totalEdges 10-ms ago, $totalEdges)
   .map(diff f.value, s.value)
   .emit($delta);

.collect($delta)
   .greatherThan('10')
   .emit($triggerVar);

.trigger($triggerVar)
   .g().compute(PageRank, $rank, [maxIterations = '10']);
```

Chapter 4

Formal semantics

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

4.1 Data model

FlowGraph builds on labeled time-evolving graphs. 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 $v.\ell.k$ ($v.\ell.k$) 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:

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

Let us 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 a point in 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'.

4.2 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.

With the evaluation model specified, we can now define the semantics of the individual constructs that compute and identify values at some point in time.

4.2.1 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

$comp : \overline{G} \to \overline{G}$

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.

4.2.2 Selection

We model a label predicate as a function p that takes in input a set of labels and returns a Boolean value

$p : \mathcal{P}(L) \to bool$

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` analogous. We model selection as a function

$sel : \overline{G} \times P \to \overline{G}$

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.

$t_{G'} = t_G$

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

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

4.2.3 Values extraction

For values extraction we refer to the `extractV` predicate, being `extractE` analogous. Let us denote KL the set of keys in labels and VL the set of values in labels. We model values extraction as a function

$extract : \overline{G} \times \mathcal{P}(KL) \to \mathcal{P}(VL)$

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.

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

4.2.4 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.

4.2.5 Definition of subgraphs

We provide the semantics of `subgraphByV`, being the definition of `subgraphByE` analogous. 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

$subgraphByV : \overline{G} \times \mathcal{P}(KL) \to \mathcal{P}(\overline{G})$

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.

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

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

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

In the following definitions, let us denote as k_G 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.

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

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, which is either a graph, or a set of groups. Then, we redefine `subgraphByV` to work on groups.

A group $Gr \in \overline{Gr}$ is either a graph $G \in \overline{G}$, or a set of 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 $n - 1$:

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

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.

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

4.2.6 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 timestamped 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.

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

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.

Chapter 5

System Implementation

FlowGraph is an open source project[1] written in Java on top of the Akka actor system[2]. Figure 5.1 depicts the architecture of FlowGraph. Similar to modern data processing platforms, it comprises a master node that coordinates many worker nodes. Clients can 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[3].

Graph vertices and edges are partitioned across worker nodes. Worker (dark grey boxes in Figure 5.1) are processes, potentially running on different machines. Each partition (light grey box in Figure 5.1) within a worker is handled by an actor. Workers can get a different number of partitions, depending on their computational and memory resources. For instance, Figure 5.1 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`.

Vertices are assigned a unique identifier upon creation. Each vertex 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 [76].

Workers store the state of the graph in main memory for improved performance. Figure 5.1 (right) expands the data structure that workers adopt to store the state of their portion of the graph: each vertex and each edge is associated with a multi-version key-value store that contains the labels of that vertex or 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.

[1] Available at `https://github.com/pietruzzo/WIP-fg`

[2] `https://doc.akka.io/docs/akka/current/general/actor-systems.html`

[3] `https://www.antlr.org`

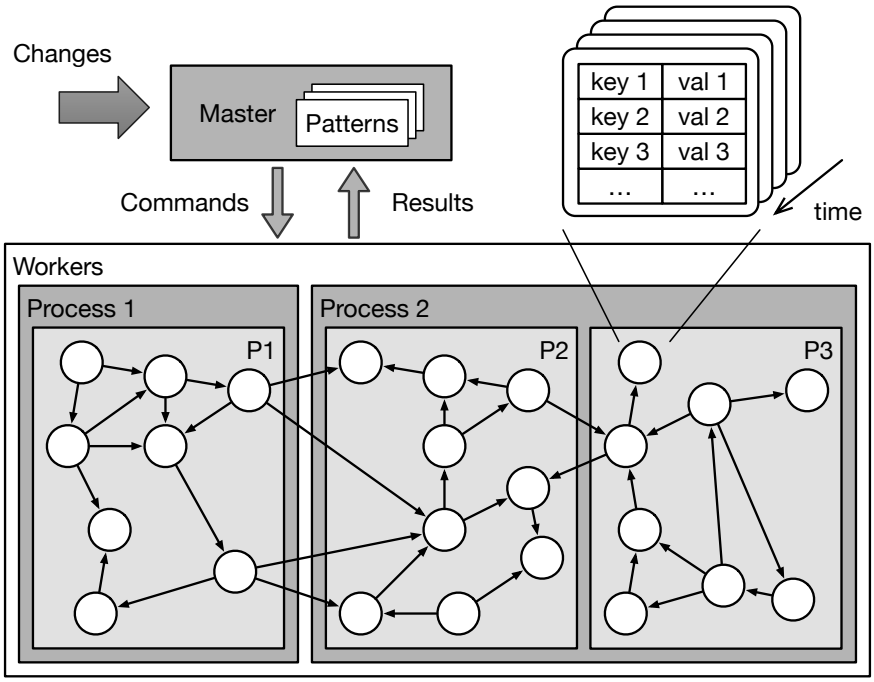

Figure 5.1: System architecture of FlowGraph.

5.1 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 aggregate information that is necessary to evaluate the pattern. Master may use these information also to skip evaluation of some pattern parts, based on aggregate values or assign-use analysis. 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 Chapter 3, 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, 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 w notifies the number of messages generated during the iteration by all

the vertices w in its partitions. The master synchronously waits for a notification from each and every worker, and requests a new iteration only in the case 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 to be used as input for further processing.

Generated set of values has an implicit information flow type depending if they refers to (i) vertex label's associated values, (ii) edge label's associated values or (iii) an aggregate set of values. It also groups set of values according to eventual defined subgraphs or grouping.

Functional operators Functional operators transform extracted values, following the same approach as modern distributed stream processing frameworks. Workers (and 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 together 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. In other words, FlowGraph dynamically infers information flow type and adapt accordingly the semantic of functional operators. Note that all operators might not be defined for all types of information flow, allowing user to partially implement functional operators interfaces.

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 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 the workers. Indeed, as explained above, workers store any value that derives from graph computations, functional, and temporal transformations. Thus, 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. Variables may be associated to vertex or edge associated label's values, aggregate values or it can correspond to a graph. As we have seen in previous paragraph, in all of this cases, variables can refers to current or a previous time instant or, if not associated to graph, it can represent a time window of values. Moreover, variables keep track of eventual sub-graphs. Underline variable type is statically inferred by FlowGraph that check the feasibility of pattern information flow.

Chapter 6

Evaluation

To be useful in practice, FlowGraph needs to detect temporal patterns that involve complex graph computations while scaling to large graphs. Accordingly, 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 the first question, we execute the page rank vertex-centric algorithm while increasing the size of the graph and the available processing resources. We show that the processing time increases linearly when moving from medium to large graphs and that FlowGraph scales linearly with the number of processors.

To answer the second question, we compare our system with GraphX [29], a state-of-the-art library for graph processing in distributed environments. GraphX builts on the Apache Spark [85] data analytics platform and is widely adopted for its efficiency and scalability. We show that FlowGraph presents comparable performance, and even outperforms GraphX with small to medium-size graphs due to a lower platform overhead.

To answer the third question, we perform detailed microbenchmarking and isolate the contributions of various pattern constructs on the performance of FlowGraph. We show that some constructs can be beneficial for processing time as they can avoid complex computations or reduce the portion of the graph they consider.

6.1 Experiment setup

We now present the setup we use throughout the entire evaluation in terms of processing infrastructure, parameters that we control during our experiments, and values that we measure.

6.1.1 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.

6.1.2 Dataset

To make sure that we measure the performance of FlowGraph when it is in a steady state, before starting any evaluation, we first load a graph into FlowGraph. 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.

6.1.3 Measured values

We designed FlowGraph to handle dynamic data, so we are primarily interested in understanding how fast it can process input data that notifies a change in the structure or content of the graph under analysis. Accordingly, our evaluation measures 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 system, and (ii) the point in time when the system 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* [38].

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. When the system runs on different machines, we report the average memory utilization of each machine.

6.1.4 Parameters

Since we are interested in evaluating FlowGraph in heterogeneous scenarios, we consider several parameters that affect its performance. We summarize them in Table 6.1, showing their default value when not differently specified.

6.2 Vertex-centric computations

As a first experiment, we focus on vertex-centric computations and we measure 1. the average processing time of FlowGraph when increasing the size of the

Parameter	Default
Size of the graph	1 M
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 6.1: Parameters used in the evaluation

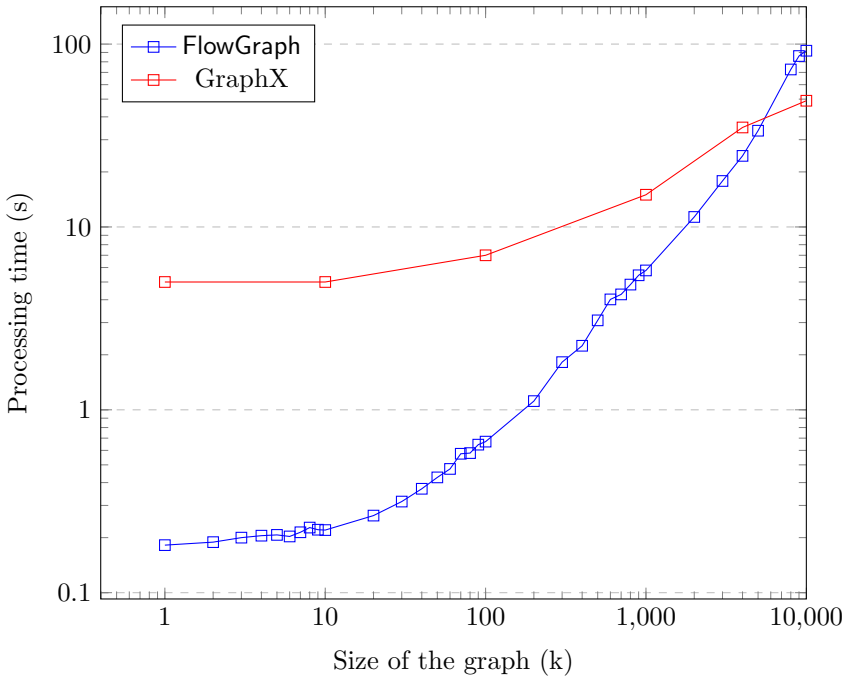

Figure 6.1: Average processing time to compute page rank (10 iterations). Comparison of FlowGraph and GraphX with increasing graph sizes.

graph, also in comparison with GraphX; 2. the scalability of FlowGraph when increasing the number of available instances.

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.

Figure 6.1 compares the average processing time of FlowGraph and GraphX while increasing the size of the graph from 1 k vertices to 10 M vertices. In absolute terms, FlowGraph performs the computation under 0.7 seconds for a graph of 100 k vertices and under 6 seconds for a graph of 1 M vertices. The processing time increases more than linearly when moving from 1 k to 100 k 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

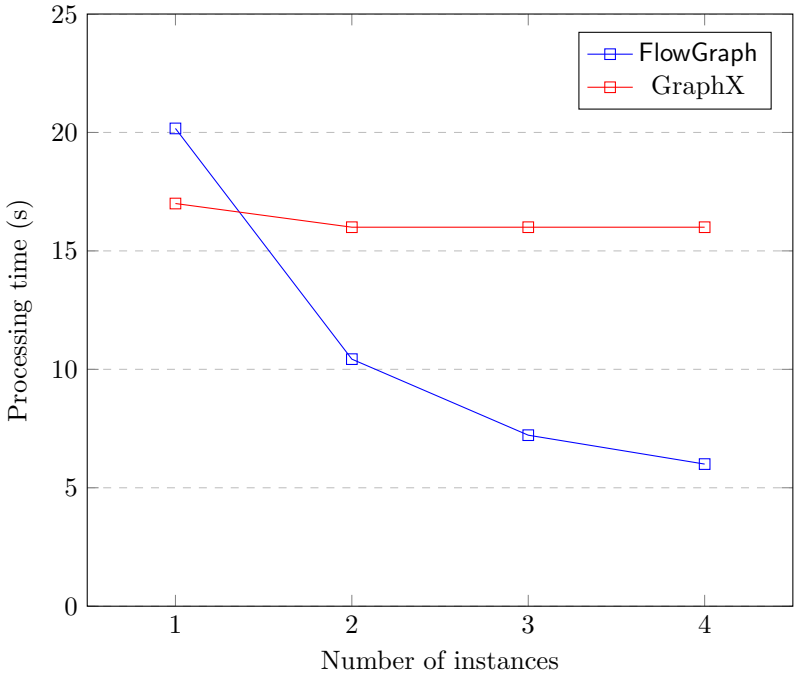

Figure 6.2: Average processing time to compute page rank (1 M vertices, 10 iterations). Comparison of FlowGraph and GraphX with increasing graph sizes.

platform is larger. In fact, FlowGraph outperforms GraphX with up to 5 M vertices, and remains comparable with 10 M vertices, despite GraphX being a mature commercial product optimized for distributed processing. While the focus of our research is temporal pattern matching in dynamic scenarios, this proves the efficiency of our prototype implementation in distributed vertex-centric computations.

Figure 6.2 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 3.4× when moving from 1 to 4 instances. In comparison, GraphX performs better 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 Figure 6.1 and in previous literature [53]. We also observed the same trend with larger graphs (up to 10 M vertices, not reported for space sake).

6.3 Pattern detection

After analyzing the performance of vertex-centric computations, we now focus on the pattern detection constructs offered in our language.

Selection To evaluate the performance of selection, we measure the average processing time to select vertices with a specific value for label `label`. Recall

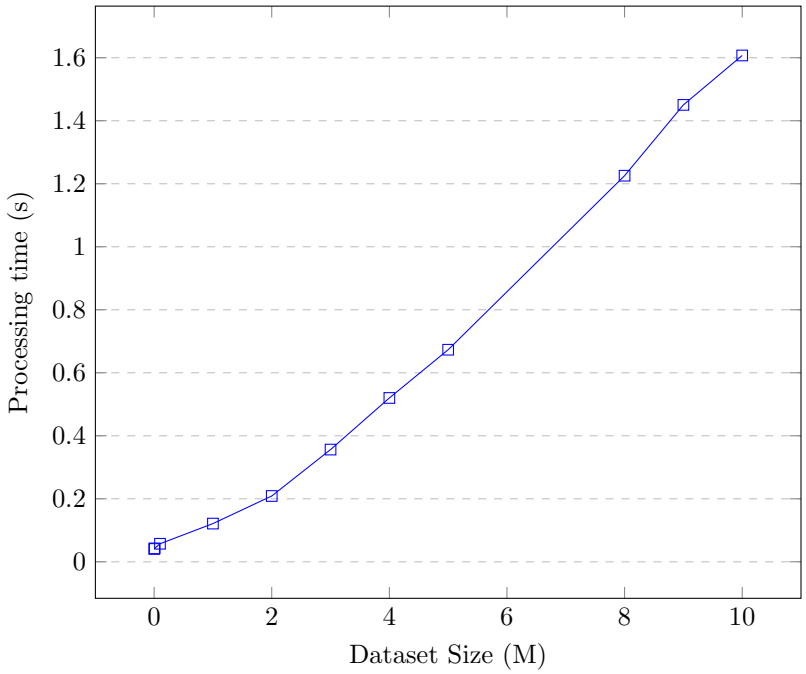

Figure 6.3: Average processing time for selection with increasing graph sizes.

that `label` gets a uniform value between 1 and 4, hence we select 25% of the vertices. Figure 6.3 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 5 M vertices in about 0.67 s, and over 10 M vertices in about 1.6 s.

6.3.1 Definition of subgraphs

Figure 6.4 shows how the performance of FlowGraph change 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. This is the same computation we presented in the previous section. `group` groups vertices according to their value of the label `label` and then performs the computation on each and every 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 1.2 seconds on a graph of 1 M vertices. However, performing the computation on smaller subgraphs rather than the entire graph reduces the `compute` time from about 6 seconds to about 1.2 seconds. The `select` operator requires about 0.2 seconds to run, but it further reduces the time to 0.382 seconds. These results confirm that FlowGraph can effectively define subgraphs and operate on them, and this has the potential to speed up vertex centric computations with respect to considering the whole graph, as we observed in the case of page rank.

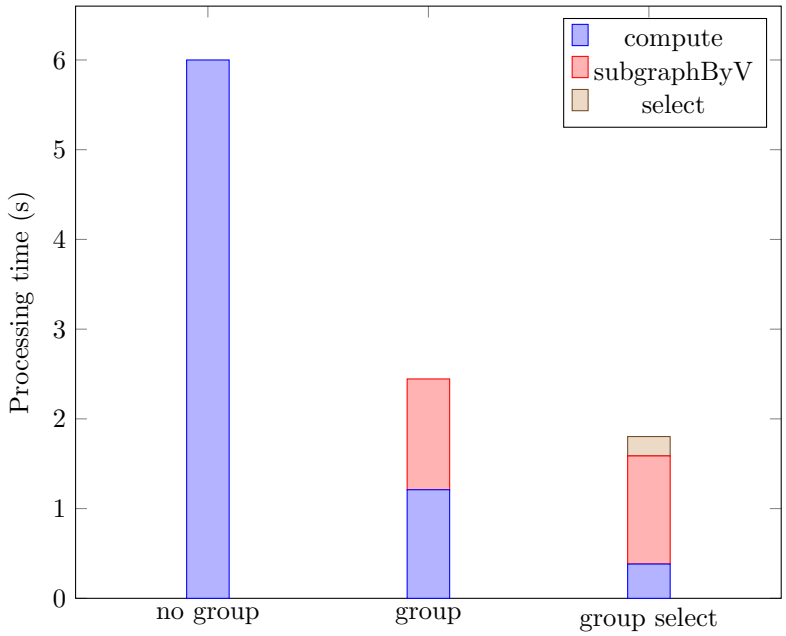

Figure 6.4: Average processing time when introducing subgraph definition and selection.

6.3.2 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. Figure 6.5 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 1000 seconds.

6.3.3 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

```
.g().compute(OutgoingEdges, $outDegree)
    .emit($graph);

.collect($outDegree).reduce(count)
   .emit($totalEdges);

.collect($totalEdges 10-ms ago, $totalEdges)
   .map(diff f.value, s.value)
   .emit($delta);
```

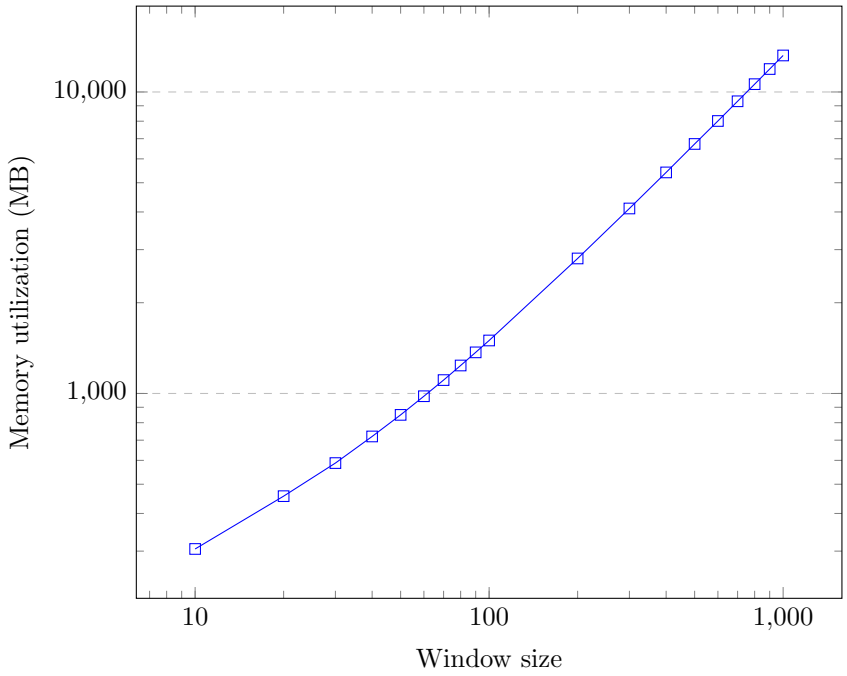

Figure 6.5: Average memory utilization per machine with increasing window size.

```
.collect($delta)
   .greatherThan('10')
   .emit($triggerVar);

.trigger($triggerVar)
   .g().compute(PageRank, $rank, [maxIterations = '10']);
```

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 second clause when the first one evaluates to false. In this case, this results in avoiding an expensive page rank computation when the number of edges has not increased significantly in the last 10 minutes.

Table 6.2 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 6.2 shows, the computation of the out degree and the evaluation of its difference over time affects performance only marginally (0.2 s on average), but the time spent to compute page rank decreases by almost 10 times, since page rank is only evaluated on

	OutDegree comput.	**OutDegree eval.**	**PageRank comput.**	**Total**
Page rank	0 s	0 s	5.98 s	5.98 s
Temporal sequence	0.10 s	0.10 s	0.60 s	0.80 s

Table 6.2: Evaluation of a temporal sequence: computing page rank at each inputput change vs computing page rank only when the number of edges has increased by at least 10 in the last 10 minutes.

10% of the input. As a consequence the average processing time decreases from 5.98 s to 0.80 s. 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.

Chapter 7

Related Work

Section 2.3 already presented the processing abstractions on which our model builds — vertex-centric computations on static graphs and logic-based CER to reason on graph evolution. In this section, we survey existing approaches that deal with dynamic graphs.

A recent survey on dynamic graph analysis [1] 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 that update the results of computations used in patterns.

The evolution analysis methods listed in the survey focus on specific problems such as community emergence and evolution [31] or shortest path distance evolution [30]. 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 have been proposed to efficiently implement evolution analysis problems in centralized or distributed settings, in batch or in near-real-time/streaming fashion [34, 68, 18, 22, 23]. 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. [71] 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* [73]— and extend it to capture a strict partial order over time when the vertices and edges that form the subgraph are added.

Graphs are also at the heart of knowledge representation in many domain, and most significantly in semantic Web. In this context, queries to the knowledge base take the form of subgraph pattern matching, as in the standard SPARQL language [59]. Although the FlowGraph model can support subgraph pattern matching using vertex-centric computations, we plan to include ad-hoc constructs and evalutation algorithms for these problems in the future, thus simplifying the definition of integrated structural and temporal patterns and making their recognition more efficient.

Several work extended SPARQL to reason on streaming graph-shaped data [11,

21, 48]. Some recent work proposed a logic framework to express and recognize temporal sequences of subgraph patterns [12, 47]. We believe that the distributed architecture presented in this paper can be beneficial in this area of application.

Finally, graph databases focus on storing and querying graph data [5, 77]. Temporal graph database exist [55], but do not address the detection of temporal patterns (in near-real-time) as we do.

Chapter 8

Conclusions and Future Works

8.1 Conclusion

This paper introduced a novel model to capture the temporal evolution of large-scale 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 presented the model semantics and its implementation in FlowGraph, a distributed system that supports large-scale dynamic graphs.

The evaluation shows the scalability of FlowGraph with respect to the graph size and the comparable performance of FlowGraph to state-of-the-art distributed frameworks for graph computations. We also think further optimizations are possible in presence of temporal patterns.

We believe that FlowGraph has the potential to open new areas of investigations in the domain of dynamic graph analysis.

8.2 Future work

Our plans for future work include (i) the implementation of fault tolerance and persistence mechanisms; (ii) to extend the library of vertex-centric algorithms, also considering incremental computations; (iii) the introduction of ad hoc constructs for subgraph pattern matching problems; (iv) to study advanced partition strategies and vertex migration approaches to reduce the cost of computation [50]; (v) the investigation of pattern-rewriting techniques [67] to optimize patterns evaluation.

Message batching and combiner FlowGraph exchanges vertex-centric computation messages in a dedicated phase at the end of superstep. One common optimization in Graph and Stream Processing is to send batches of messages as soon as they reach a given size threshold [46, 85]. Through this mechanism, it has been observed a more efficient use of network bandwidth and a lowering in response time.

Some frameworks also implements a *combiner* in order to de-duplicate repeated messages as we have seen in Section 2.1.4.

We believe that the integration of these solutions may lower the slope of response time curve in Figure 6.1 paying a small overhead.

Fault Tolerance and Persistence FlowGraph has not a fully developed strategy to handle faults and guarantee data persistence in the distributed environment, but we also think that defining strategies to detect and recovery from fails is critical for applications that FlowGraph propose to address.

Communication is based on Akka Actors system middleware [32] that already guarantee *at-most-once* delivery and message ordering. Akka has also mechanisms to detect failed remote actors. FlowGraph has been develop also with an acknowledge mechanism for each message exchanged. Therefore, we should be able to detect any system fault both on machines and channels, but we still need to define a strategy for data replication, distributed persistence and recovery.

Similar graph processing frameworks employs check-pointing strategies [46], distributed in-memory databases as Facebook's *RocksDB* or *Redis* and distributed file-systems such as *Hadoop Distributed File System* (HDFS). In particular, with above mentioned in-memory databases, we observed $\tilde{1}$-2 orders of magnitude of performance degradation in random access to stored information with respect to heap random access. On the other hand, we noticed only a $\tilde{5}$ time degradation in sequential information retrieval of large chunks. Thus, a trade of checkpoint granularity should be carefully tuned.

Incremental computation The intuitive idea behind incremental computations is that minor incremental changes of inputs may not require to reprocess the entire work but only the affected portion. Restricting the scope of computation to affected vertices should allow to generate less messages as well as converge in less supersteps [43].

Advanced partitioning strategies FlowGraph support only graph partitioning based on the hashing of vertex-id. This partition strategy allows to obtain a relatively balanced set of partitions in terms of number of vertices per partition. However, this strategy could impose an high number of edge cuts, underfitting the locality properties proper of graphs.

We have presented, in Section 2.1.4, some algorithms and techniques to minimize vertex cuts, balance computation's execution and we cited some studies that propose to migrate vertices from one partition to another in order to rebalance a distributed graph.

FlowGraph may be also easily extended to support also *partition-based* computations taking advantage of specific partitioning techniques (Section 14).

Pattern Rewriting As seen in Chapter 5, FlowGraph is already able to recognize definition-usage dependencies of variables, also trimming pattern chain executions based on results of other chains. However, execution model is still limited by a sequential interpretation of pattern with few static optimization at parsing time, leaving to user the charge of writing optimized versions of the same pattern. A future work could consist in exploring different static (parsing

time) and dynamic (profiling) strategies for operation reordering, decoupling, postponing and trimming based on their dataflow.

Acknowledgement

I would like to express my gratitude to my supervisor Alessandro Margara for the exceptional guidance, deep knowledge about the topic, useful comments, remarks and engagement through the learning process of this master thesis. Furthermore, I would like to thank professor Matteo Rossi for his precious suggestions, competence and collaboration. I would also like to give thanks to Hassan Nazeer Chaudhry for introducing me to the topic, for the amount of time he invested in our meetings as well as for the valuable support on the way.

Lastly, I would like to thank my loved ones, especially Chiara, my family, my cousins and also who left us recently. They have given me constant support and love throughout the entire process. I will be grateful forever for your love.

Bibliography

[1] Charu Aggarwal and Karthik Subbian. Evolutionary network analysis: A survey. *ACM Computing Surveys*, 47(1):10:1–10:36, 2014.

[2] Marcos K. Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing*, page 53–61, New York, NY, USA, 1999. Association for Computing Machinery.

[3] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric 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. *Proc. VLDB Endow.*, 8(12):1792–1803, 2015.

[4] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, page 120–124, New York, NY, USA, 2004. Association for Computing Machinery.

[5] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys*, 40(1):1:1–1:39, 2008.

[6] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394, 2015.

[7] A. Artikis, M. Sergot, and G. Paliouras. An event calculus for event recognition. *IEEE Transactions on Knowledge and Data Engineering*, 27(4):895–908, 2015.

[8] Alexander Artikis, Alessandro Margara, Martin Ugarte, Stijn Vansummeren, and Matthias Weidlich. Complex event recognition languages: Tutorial. In *Proceedings of the International Conference on Distributed and Event-based Systems*, DEBS '17, pages 7–10, New York, NY, USA, 2017. ACM.

[9] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of*

the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS '02, page 1–16, New York, NY, USA, 2002. Association for Computing Machinery.

[10] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *SIGMOD Rec.*, 30(3):109–120, September 2001.

[11] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. C-sparql: Sparql for continuous querying. In *Proceedings of the International Conference on World Wide Web*, WWW '09, pages 1061–1062, New York, NY, USA, 2009. ACM.

[12] Harald Beck, Minh Dao-Tran, Thomas Eiter, and Michael Fink. Lars: A logic-based framework for analyzing reasoning over streams. In *Proceedings of the AAAI Conference on Artificial Intelligence*, AAAI'15, page 1431–1438. AAAI Press, 2015.

[13] Dhruba Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 11(2007):21, 2007.

[14] Lars Brenna, Alan Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, and Walker White. Cayuga: A high-performance event processing engine. In *Proceedings of the International Conference on Management of Data*, SIGMOD '07, pages 1100–1102, New York, NY, USA, 2007. ACM.

[15] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: Efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1–2):285–296, September 2010.

[16] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *IEEE Data Engineering Bullettin*, 38(4):28–38, 2015.

[17] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, 2015.

[18] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the European Conference on Computer Systems*, EuroSys '12, pages 85–98, New York, NY, USA, 2012. ACM.

[19] Gianpaolo Cugola and Alessandro Margara. Tesla: A formally defined event specification language. In *Proceedings of the International Conference on Distributed Event-Based Systems*, DEBS '10, pages 50–61, New York, NY, USA, 2010. ACM.

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

[21] Emanuele Della Valle, Stefan Schlobach, Markus Krötzsch, Alessandro Bozzon, Stefano Ceri, and Ian Horrocks. Order matters! harnessing a world of orderings for reasoning over massive data. *Semantic Web*, 4(2):219–231, 2013.

[22] Benjamin Erb, Dominik Meissner, Jakob Pietron, and Frank Kargl. Chronograph: A distributed processing platform for online and batch computations on event-sourced graphs. In *Proceedings of the International Conference on Distributed and Event-based Systems*, DEBS '17, pages 78–87, New York, NY, USA, 2017. ACM.

[23] Benjamin Erb, Dominik Meiundefinedner, Frank Kargl, Benjamin A. Steer, Felix Cuadrado, Domagoj Margan, and Peter Pietzuch. Graphtides: A framework for evaluating stream-based graph processing platforms. In *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, GRADES-NDA '18, New York, NY, USA, 2018. ACM.

[24] Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Publications Co., 1st edition, 2010.

[25] Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl. Spinning fast iterative data flows. *arXiv preprint arXiv:1208.0088*, 2012.

[26] Alexandros V. Gerbessiotis and Leslie G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22(2):251–267, 1994.

[27] Nikos Giatrakos, Elias Alevizos, Alexander Artikis, Antonios Deligiannakis, and Minos Garofalakis. Complex event recognition in the big data era: a survey. *The VLDB Journal*, Jul 2019.

[28] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Hollywood, CA, 2012. USENIX Association.

[29] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 599–613, USA, 2014. USENIX Association.

[30] Manish Gupta, Charu C. Aggarwal, and Jiawei Han. Finding top-k shortest path distance changes in an evolutionary network. In *Proceedings of the International Conference on Advances in Spatial and Temporal Databases*, SSTD '11, pages 130–148. Springer, 2011.

[31] Manish Gupta, Charu C. Aggarwal, Jiawei Han, and Yizhou Sun. Evolutionary clustering and analysis of bibliographic networks. In *Proceedings of the International Conference on Advances in Social Networks Analysis and Mining*, ASONAM '11, pages 63–70. IEEE, 2011.

[32] Munish Gupta. *Akka essentials*. Packt Publishing Ltd, 2012.

[33] Minyang Han and Khuzaima Daudjee. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proceedings of VLDB Endow.*, 8(9):950–961, 2015.

[34] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: A graph engine for temporal graph analysis. In *Proceedings of the European Conference on Computer Systems*, EuroSys '14, pages 1:1–1:14, New York, NY, USA, 2014. ACM.

[35] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. Time-evolving graph processing at scale. In *Proceedings of the International Workshop on Graph Data Management Experiences and Systems*, GRADES '16, pages 5:1–5:6, New York, NY, USA, 2016. ACM.

[36] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, March 2004.

[37] V. Kalavri, V. Vlassov, and S. Haridi. High-level programming abstractions for distributed graph processing. *IEEE Transactions on Knowledge and Data Engineering*, 30(2):305–324, Feb 2018.

[38] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl. Benchmarking distributed stream data processing systems. In *Proceedings of the International Conference on Data Engineering*, ICDE'18, pages 1507–1518. IEEE, 2018.

[39] G. Karypis and V. Kumar. Parallel multilevel graph partitioning. In *Proceedings of International Conference on Parallel Processing*, pages 314–319, 1996.

[40] Taiwo Kolajo, Olawande Daramola, and Ayodele Adebiyi. Big data stream analysis: a systematic literature review. *Journal of Big Data*, 6, Jun 2019.

[41] Paul Kosinski. A data flow language for operating systems programming. *ACM SIGPLAN Notices*, 8:89–94, 09 1973.

[42] Ben Lee and Ali R Hurson. Dataflow architectures and multithreading. *Computer*, 27(8):27–39, 1994.

[43] Qiang Liu, Xiaoshe Dong, Heng Chen, and Yinfeng Wang. Incpregel: an incremental graph parallel computation model. *Frontiers of Computer Science*, 12(6):1076–1089, 2018.

[44] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proceedings of VLDB Endow.*, 5(8):716–727, 2012.

[45] Andrew Lumsdaine, Doublas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in Parallel Graph Processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[46] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[47] Alessandro Margara, Gianpaolo Cugola, Dario Collavini, and Daniele Dell'Aglio. Efficient temporal reasoning on streams of events with dotr. In *Proceedings of the Extended Semantic Web Conference*, ESWC '18, pages 384–399. Springer International Publishing, 2018.

[48] Alessandro Margara, Jacopo Urbani, Frank van Harmelen, and Henri Bal. Streaming the web. *Web Semantics*, 25(C):24–44, March 2014.

[49] Nathan Marz and James Warren. *Big Data: Principles and best practices of scalable realtime data systems.* Manning Publications Co., 2015.

[50] C. Mayer, M. A. Tariq, R. Mayer, and K. Rothermel. Graph: Traffic-aware graph processing. *IEEE Transactions on Parallel and Distributed Systems*, 29(6):1289–1302, June 2018.

[51] Dennis McCarthy and Umeshwar Dayal. The architecture of an active database management system. *SIGMOD Rec.*, 18(2):215–224, June 1989.

[52] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys*, 48(2):25:1–25:39, 2015.

[53] Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! but at what cost? In *Proceedings of the USENIX Conference on Hot Topics in Operating Systems*, HOTOS '15, pages 14:1–14:6. USENIX Association, 2015.

[54] Yuan Mei and Samuel Madden. Zstream: A cost-based query processor for adaptively detecting composite events. In *Proceedings of the International Conference on Management of Data*, SIGMOD '09, pages 193–206, New York, NY, USA, 2009. ACM.

[55] Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen. Immortalgraph: A system for storage and analysis of temporal graphs. *Transactions on Storage*, 11(3):14:1–14:34, 2015.

[56] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455, 2013.

[57] T. Suzumura N. T. Bao. Towards highly scalable pregel based graph processing platform with x10. *Proc. 22nd Int.*, page 501–508, 2013.

[58] Norman Paton and Oscar Díaz. Active database systems. *ACM Comput. Surv.*, 31:63–103, 03 1999.

[59] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. *ACM Transactions on Database Systems*, 34(3):16:1–16:45, 2009.

[60] William Cohen Philip Stutz, Abraham Bernstein. Signal/collect: Graph algorithms for the (semantic) web. In Patel-Schneider P.F. et al., editor, *The Semantic Web*, 2010.

[61] Lu Qin, Jeffrey Xu Yu, Lijun Chang, Hong Cheng, Chengqi Zhang, and Xuemin Lin. Scalable big graph processing in mapreduce. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 827–838, New York, NY, USA, 2014. Association for Computing Machinery.

[62] Fatemeh Rahimian, Amir H. Payberah, Sarunas Girdzijauskas, Mark Jelasity, and Seif Haridi. A distributed algorithm for large-scale graph partitioning. *ACM Transactions on Autonomous and Adaptive Systems*, 10:1–24, 06 2015.

[63] Marko A. Rodriguez. The gremlin graph traversal machine and language (invited talk). *Proceedings of the 15th Symposium on Database Programming Languages - DBPL 2015*, 2015.

[64] David S Rosenblum and Alexander L Wolf. A design framework for internet-scale event observation and notification. In *Proceedings of the 6th European SOFTWARE ENGINEERING conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 344–360, 1997.

[65] Semih Salihoglu and Jennifer Widom. Optimizing graph algorithms on pregel-like systems. *Proc. VLDB Endow.*, 7(7):577–588, March 2014.

[66] Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M. Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 979–990, New York, NY, USA, 2014. Association for Computing Machinery.

[67] Nicholas Poul Schultz-Moller, Matteo Migliavacca, and Peter Pietzuch. Distributed complex event processing with query rewriting. In *Proceedings of the International Conference on Distributed Event-Based Systems*, DEBS '09, New York, NY, USA, 2009. ACM.

[68] Dipanjan Sengupta and Shuaiwen Leon Song. Evograph: On-the-fly efficient mining of evolving graphs on gpu. In Julian M. Kunkel, Rio Yokota, Pavan Balaji, and David Keyes, editors, *High Performance Computing*, ISC '17, pages 97–119. Springer, 2017.

[69] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 505–516, New York, NY, USA, 2013. Association for Computing Machinery.

[70] Yogesh Simmhan, Alok Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi Raghavendra, and Viktor Prasanna. Goffish: A sub-graph centric framework for large-scale graph analytics. In *Euro-Par Parallel Processing*, pages 451–462. Springer, 2014.

[71] Chunyao Song, Tingjian Ge, Cindy Chen, and Jie Wang. Event pattern matching over graph streams. *Proceedings of VLDB Endow.*, 8(4):413–424, 2014.

[72] Utkarsh Srivastava and Jennifer Widom. Flexible time management in data stream systems. In *Proceedings of the Symp. on Principles of Database Systems*, PODS '04, pages 263–274, New York, NY, USA, 2004. ACM.

[73] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. Efficient subgraph matching on billion node graphs. *Proceedings of VLDB Endow.*, 5(9):788–799, 2012.

[74] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: A system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 425–440, New York, NY, USA, 2015. Association for Computing Machinery.

[75] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[76] Luis M. Vaquero, Felix Cuadrado, Dionysios Logothetis, and Claudio Martella. Adaptive partitioning for large-scale dynamic graphs. In *Proceedings of the International Conference on Distributed Computing Systems*, ICDCS '14, pages 144–153. IEEE, 2014.

[77] Aleksa Vukotic, Nicki Watt, Tareq Abedrabbo, Dominic Fox, and Jonas Partner. *Neo4j in Action*. Manning Publications Co., USA, 2014.

[78] Simon F Wail. Can dataflow machines be programmed with an imperative language. *Advanced Topics in Dataflow Computing and Multithreading*, pages 229–265, 1995.

[79] Guozhang Wang, Wenlei Xie, Alan J Demers, and Johannes Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, volume 13, pages 3–6, 2013.

[80] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *Proceedings of the International Conference on Management of Data*, SIGMOD '06, pages 407–418, New York, NY, USA, 2006. ACM.

[81] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. Sync or async: Time to fuse for distributed graph-parallel computation. *SIGPLAN Not.*, 50(8):194–204, January 2015.

[82] Reynold S Xin, Josh Rosen, Matei Zaharia, Michael J Franklin, Scott Shenker, and Ion Stoica. Shark: Sql and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*, pages 13–24, 2013.

[83] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28. USENIX, 2012.

[84] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[85] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.

[86] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, page 56–65, 2016.