Percolator: Scalable Pattern Discovery in Dynamic Graphs

Sutanay Choudhury
Pacific Northwest National Laboratory, USA
sutanay.choudhury@pnnl.gov

Sumit Purohit
Pacific Northwest National Laboratory, USA
sumit.purohit@pnnl.gov

Peng Lin
Washington State University, USA
plin1@eecs.wsu.edu

Yinghui Wu
Washington State University, USA
Pacific Northwest National Laboratory, USA
yinghui@eecs.wsu.edu

Lawrence Holder
Washington State University, USA
holder@eecs.wsu.edu

Khushbu Agarwal
Pacific Northwest National Laboratory, USA
khushbu.agarwal@pnnl.gov

ABSTRACT

We demonstrate Percolator, a distributed system for graph pattern discovery in dynamic graphs. In contrast to conventional mining systems, Percolator advocates efficient pattern mining schemes that (1) support pattern detection with keywords; (2) integrate incremental and parallel pattern mining; and (3) support analytical queries such as trend analysis. The core idea of Percolator is to dynamically decide and verify a small fraction of patterns and their instances that must be inspected in response to buffered updates in dynamic graphs, with a total mining cost independent of graph size. We demonstrate a) the feasibility of incremental pattern mining by walking through each component of Percolator, b) the efficiency and scalability of Percolator over the sheer size of real-world dynamic graphs, and c) how the user-friendly GUI of Percolator interacts with users to support keyword-based queries that detect, browse and inspect trending patterns. We demonstrate how our system effectively supports event and trend analysis in social media streams and research publication, respectively.

CCS CONCEPTS

• Information systems → Data stream mining; **• Theory of computation → Dynamic graph algorithms**;

KEYWORDS

Graph mining; parallel system; data stream

ACM Reference Format:
Sutanay Choudhury, Sumit Purohit, Peng Lin, Yinghui Wu, Lawrence Holder, and Khushbu Agarwal. 2018. Percolator: Scalable Pattern Discovery in Dynamic Graphs. In *Proceedings of WSDM 2018: The Eleventh ACM International Conference on Web Search and Data Mining (WSDM 2018).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3159652.3160589

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
WSDM 2018, February 5–9, 2018, Marina Del Rey, CA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5581-0/18/02…$15.00
https://doi.org/10.1145/3159652.3160589

1 INTRODUCTION

Discovering emerging events from massive dynamic data is a critical need in a wide range of applications. Real-world events in dynamic networks (*e.g.,* Web, social media and cyber networks) are often represented as *graph patterns*. Although desirable, discovering such patterns is more challenging than its counterpart over item streams [2]. It requires effective querying and mining over graphs that bear constant changes, while pattern mining is already expensive over static graphs [5, 8]. Moreover, it is hard to reduce the computational complexity (NP-hard). Pattern discovery should also support keyword input, *i.e.,* to discover and track patterns relevant to user-specified keywords.

One approach is to leverage incremental computation that has been applied to update query results [6]. The idea is to dynamically identify a fraction of data that must be inspected to update the patterns in response to changes. Another method is to develop parallel mining [7] to cope with the sheer amount of data. *Can we combine parallel and incremental computation to support scalable pattern discovery in massive dynamic graphs?*

Percolator. This motivates us to develop Percolator, a prototype system that combines both incremental mining and parallel mining for feasible pattern detection over graph streams. It differs from conventional systems with the following unique new features.

(1) Keyword specified patterns. Percolator supports the discovery of (general) graph patterns that pertain to user-specified keywords. It finds informative and concise patterns characterized by maximal patterns and their activeness measures. It also supports both ad-hoc top pattern detection and offline trend analysis.

(2) Incremental mining. Percolator supports *incremental* pattern mining to avoid rediscover patterns from scratch upon receiving changes. Given discovered patterns Σ over a graph, and a batch of edge updates, it incrementally updates Σ by automatically tracking and only re-verifying a set of patterns and matches. This avoids unnecessary computation from scratch, with a cost only determined by the changes of results and data, but independent of the graph size.

(3) Scale-up. Percolator is a parallel system implemented on top of Apache Spark. It parallelizes the incremental graph mining by dynamically constructing and exchanging messages that contain affected patterns and triples needed for incremental verification, in parallel and only when necessary. This reduces communication cost and ensures the scalability.

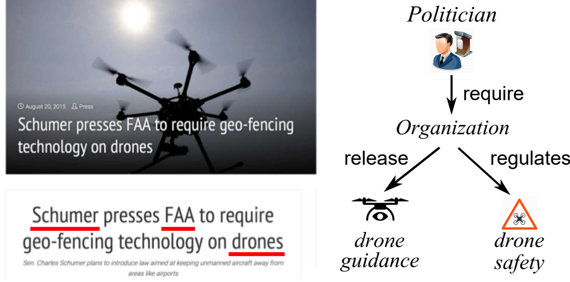

Figure 1: Emerging event pattern from News data as triples

(4) Easy-to-use. The Percolator system is easy to use. It provides a user-friendly GUI, a built-in natural-language query constructor to support keyword-based pattern discovery, a visualization component to inspect and interpret the results.

Demo overview. We next demonstrate Percolator as follows. (1) We walk through each component of Percolator from pattern models to incremental and parallel mining, to demonstrate the feasibility of Percolator over massive dynamic graphs. (2) We demonstrate the applications of Percolator with real-world event analysis scenarios in social media (news) and academic data. We demonstrate its query interface, performance analysis and visual analysis interface to demonstrate emerging events and their text interpretations.

2 PATTERN AND GRAPH STREAM MODELS

We start with the graph streams and the pattern model in Percolator.

Dynamic Graph. Percolator models a dynamic graph $\mathcal{G}_T$ as a set of triples with timestamps. Each triple e=$<s, p, o>$ consists of a subject s, predicate p (a relation) and an object o, and each of s, p and o has a label (*e.g.,* URI). (1) A snapshot of $\mathcal{G}_T$ at time i is a graph G_i induced by triples at time i. (2) At each timestamp i, a batch of triples ΔE updates snapshot G_i to G_{i+1}.

Patterns. A pattern P is a labeled connected graph $(V_p, E_p, \bar{u})$ with a set of labeled pattern nodes V_p and a set of labeled pattern edges $E_p \subseteq V_p \times V_p$. Given a set of keywords $\mathcal{K}$=$\{k_1, \ldots k_n\}$, a pattern P pertains to $\mathcal{K}$ if for every keyword k_i $i \in [1, n]$, there exists a keyword node u_i in P such that $L(u_i)$=k_i.

Activeness. By default, Percolator characterizes active patterns with *minimum image support* [1]. Given a snapshot G_i, a pattern P and a set of (user-specified) keyword nodes $\bar{u}$ in P, the activeness of P in G_i ($\text{Act}(P, G_i)$) is quantified as min $|\{M(u)|u \in \bar{u}\}|$, where $M(u)$ refers to the matches of node u induced by all the subgraph isomorphisms from P to G_i, and u ranges over $\bar{u}$. Given an activeness threshold θ, P is active in G_i *w.r.t.* θ if $\text{Act}(P, G_i) \geq \theta$. We remark that the activeness of P preserves *anti-monotonicity* [1] *w.r.t.* a fixed $\bar{u}$: at any time i, $\text{Act}(P, G_i) \leq \text{Act}(P', G_i)$ *w.r.t.* a fixed set of keyword nodes, if P' is obtained by adding edges to P.

Maximal patterns. We are interested in detecting "informative" patterns as maximal ones. Indeed, small patterns may be active, but usually tend to be trivial ones. A pattern P is maximal in G_i *w.r.t.* threshold θ, if (1) P is active *w.r.t.* θ, and (2) there exists no active super-pattern P' of P. Percolator discovers maximal patterns that pertain to a set of user-specified keywords $\mathcal{K}$.

Example 1: Figure 1 illustrates an active pattern detected and tracked by Percolator, specified by keywords "*Politician*", "*drone*" and "*organization*". It is discovered in a dynamic graph (RDF triples) extracted from news articles by Nous [3], a knowledge graph construction engine. The pattern and its matches reveal events regarding emerging concerns of drone safety. It verifies that *politicians* (*e.g.,* "Schumer") are pressing *organizations* (*e.g.,* "Federal Aviation Administration (FAA)") to regulate *drones* and provide *guidance*.

Pattern discovery in dynamic graphs. Given a dynamic graph $\mathcal{G}_T$, a set of keywords $\mathcal{K}$, a set of active patterns Σ pertain to $\mathcal{K}$, (obtained by"from-scratch" discovery), Percolator updates Σ in response to a set of edge transactions ΔE applied to G, without rediscovery Σ from scratch, and outputs updated Σ upon request.

3 FOUNDATIONS OF PERCOLATOR

The Percolator system is built on two principles: *incremental pattern mining* and *parallel mining*. (1) Instead of discovering patterns from scratch, each time $\mathcal{G}_T$ is updated, it performs necessary computation that suffices to update Σ (Section 3.1). (2) To scale the process over large $\mathcal{G}_T$, it performs incremental mining in parallel, and aggregates the changes to update Σ (Section 3.2).

3.1 Incremental Discovery

Percolator dynamically identifies and only verifies a set of "affected" patterns that must be inspected in order to update Σ. This novel principle is implemented by three core components: *Stream Manager*, *Affected Pattern Detector* and *Incremental Verifier*.

Stream manager. The stream manager of Percolator processes $\mathcal{G}_T$ as a triple stream and manages the built-in structures below.

Triple Buffer. Percolator uses a buffer B with a tunable size to cache the edge updates in *batches*. (1) It caches the updates ΔE in multiple batches bounded by the buffer size. It also maintains a buffer map $B.M$, which points each triple $e \in B$ to a single-edge pattern $B.M(e)$ having e as a match. A pattern P is "hit" by e if P contains a pattern edge $B.M(e)$. (2) Percolator applies a novel *load shedding* strategy to prune triples in B: only triples with one end node having a keyword label is cached for processing. Indeed, only these triples may affect the activeness of patterns by the definition of activeness. All others are applied to G directly without further processing.

Affected Pattern Detector. The manager maintains the active events with a lattice $\mathcal{T}$ as commonly used in constrained graph mining. In addition, it tracks the activeness of the patterns. Upon receiving a batch of triples $\Delta E_B \subseteq \Delta E$, the *affected pattern detector* of Percolator interacts with incremental verifier (to be discussed) and dynamically identifies a "minimal" set $\mathcal{P}$ of patterns that are necessary to be inspected to update Σ. (1) It initializes $\mathcal{P}$ as the patterns "hit" by $e.M$, and sends $\mathcal{P}$ to the incremental verifier to update their activeness. (2) For each verified pattern $P \in \mathcal{P}$, it "propagates" $\mathcal{P}$ by taking two actions below.

Downward propagation: If $\text{Act}(P, G_i) \geq \theta$, it updates $\mathcal{P}$ as:

$$\mathcal{P} := \mathcal{P} \cup P^+,$$

where P^+ refers to the patterns obtained by adding an edge to P, *i.e.,* the possible "children" of P in $\mathcal{T}$. That is, it simulates the exploration of larger patterns in $\mathcal{T}$ as P remains to be active.

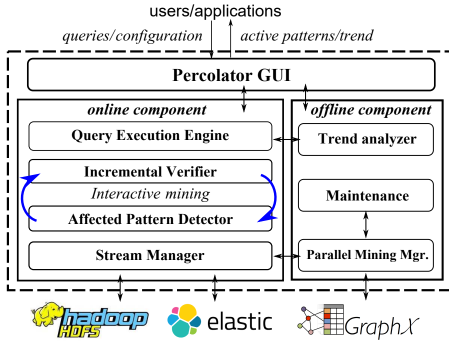

Figure 2: Architecture of Percolator

Upward propagation: If $\mathrm{Act}(P, G_i) < \theta$ (*i.e.,* becomes inactive due to *e.g.,* decrease in edge frequency), it updates $\mathcal{P}$ as:

$$\mathcal{P} := \mathcal{P} \cup P^-,$$

where P^- refers to the "parents" of P in $\mathcal{T}$, obtained by removing an edge from P. That is, it explores smaller patterns that may become new maximal ones as P becomes inactive.

Note that both P^+ and P^- (including new patterns) can be constructed without reconstructing $\mathcal{T}$ from scratch. The detector stops downward (resp. upward) propagation at pattern P if P^+ (resp. P^-) is $\emptyset$, and checks whether no child of P is active. If so, it inserts P to Σ as a newly discovered maximal pattern.

Incremental Verifier. The *Incremental Verifier* of Percolator updates the activeness of each pattern $P \in \mathcal{P}$. If P is affected, it updates $\mathrm{Act}(P, G_i)$ by "incrementalizing" the conventional subgraph isomorphism test, which processes single-edge patterns in $\mathcal{T}$ hit by $B.M$ and "percolate-up" to P (see below). (2) Otherwise, P is obtained by either upward or downward propagation, and $\mathrm{Act}(P, G_i)$ remains to be the same.

Percolate-up. The Incremental Verifier adopts a "*percolate-up*" strategy to reduce the processing cost of updates. Given ΔE^B, it partitions ΔE^B into groups by consulting the buffer map $B.M$, where each group hits a single-edge pattern P_0. For each pattern $P \in \mathcal{P}$, it then evaluates a sequence of patterns $\{P_0, \ldots, P_j\}$ sorted by their size, such that P_i is a sub-pattern of P_{i+1} ($i \in [0, j-1]$), and P_j=P. That is, it "*percolates*" the edges up against the affected patterns, from smaller ones to larger ones. Moreover, it never re-evaluates a pattern P_i if P_i is verified (in other sequences). This effectively reduces redundant verifications (especially for "overlapping" patterns that share sub-patterns), and early terminates at inactive patterns, guaranteed by the anti-monotonicity of the activeness.

Upon each batch of updates, Percolator interleaves Affected Pattern Detector and Incremental Verifier to "lazily" perform necessary amount of verification. The interaction repeats until no pattern can be added to $\mathcal{P}$. It then reports updated Σ upon request.

3.2 Parallel mining

To cope with large $\mathcal{G}_T$, Percolator parallelizes the incremental mining over a set of distributed, shared-nothing workers.

Parallel mining manager. This component executes the parallel computation of Percolator. It maintains the following. (1) A fragmentation $\mathcal{F}$ of dynamic graph $\mathcal{G}_T$ is a partition of the snapshot G over n workers $\{F_1, \ldots, F_n\}$, where each worker W_j manages a subgraph G_j of G. By default, Percolator applies balanced 2D partition. (2) The batch updates ΔE_i is fragmented as $\{\Delta E_1, \ldots, \Delta E_n\}$, where each ΔE_j changes F_i to $F_i \oplus \Delta E_j$, respectively. (3) The pattern lattice $\mathcal{T}$ is synchronized among all the workers.

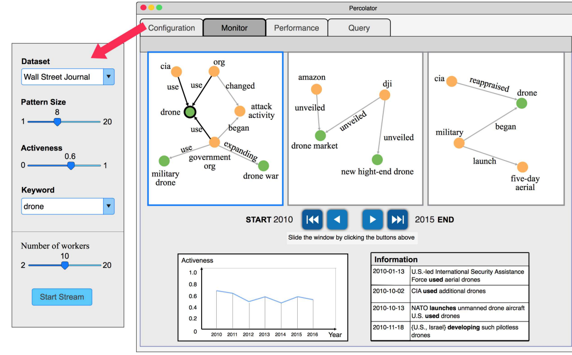

Figure 3: Percolator GUI

Parallel mining. Given $\mathcal{F}$ and a batch of updates ΔE, Percolator "parallelizes" the sequential incremental mining (Section 3.1) following a Bulk Synchronous Parallel model, and runs in supersteps.

(1) Upon receiving ΔE_j, each worker W_j invokes Affected Pattern Detector to identify a local set of affected patterns $\mathcal{P}_j$ due to local changes of F_j, and invokes Incremental Verifier to update their local activeness, in parallel. For each pattern P with diameter d that cannot be verified locally, it extends F_j with d-hop neighbors of the "border" nodes of F_j in G and performs local verification.

(2) Once all the workers complete the local verification, the coordinator W_o computes affected patterns $\mathcal{P}$=$\bigcup_{j \in [1,n]} \mathcal{P}_j$, assembles the local activeness of each pattern in $\mathcal{P}$, and updates Σ when necessary. It then broadcasts $\mathcal{P}$ to all workers.

The above two steps repeat until no new affected patterns can be added to $\mathcal{P}$, and all the affected patterns are verified. Percolator then returns Σ updated at the coordinator W_o.

4 SYSTEM OVERVIEW

Architecture. Percolator consists of three components (Figure 2).

Online pattern discovery: consists of four modules, including *Stream Manager*, *Affected Pattern Detector*, *Incremental Verification* (Section 3.1), and a *query execution engine* to evaluate ad-hoc queries.

Offline pattern analysis: consists of three modules, including a *trend analyzer* to support trend analysis, a *maintenance component* to synchronize the changes to underlying graphs, and the *parallel mining manager* to manage the distributed environment, *e.g.,* the parallel configuration, data partitioning and fault-tolerance.

GUI. The user-friendly Percolator GUI (illustrated in Figure 3) contains a *configuration panel* to receive configurations (*e.g.,* activeness threshold, number of workers). Users can browse and inspect active patterns in the *monitor panel*, which includes the activeness curve, and the details of their matches. The *performance panel* (not shown) visualizes performance analysis. Finally, a built-in *Query panel* allows users to issue natural-language style queries, supported by built-in query parsers.

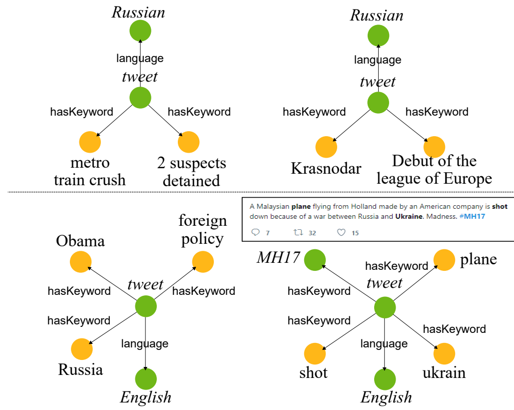

Figure 4: Top-k Twitter Patterns

Implementation. Percolator is implemented in Scala, and built on top of Apache Spark and HDFS with core functions in GraphX library. The graph stream is represented as distributed arrays (RDDs) managed by Spark. To support fast stream access, it uses in-memory Elasticsearch[1] to manage the intermediate results (*e.g.,* the mapping in Stream Manager) as key-value pairs. Percolator utilizes our prior systems for graph construction [3] and querying [4] as input and verification interfaces, respectively.

5 DEMONSTRATION OVERVIEW

The target audience of the demo includes anyone who is interested in understanding complex events and trends over data streams. Our system[2] is deployed on a cluster of 16 nodes (with one serving as the coordinator). Each node is equipped with an Intel Xeon processor (2.3 GHz) with 16 cores and 64 GB memory.

Settings. We use the following settings.

Datasets. Our real world datasets include: (1) **Twitter**, a collection of dynamic knowledge graphs with in total 5 million triples and in batches of 1.5 million triples per day. (2) **MAG**, a citation network with 153.6 million triples. Each batch of triples contains 8 million nodes and 22 million edges in one year window.

Ad-hoc queries. We invite users to inspect patterns discovered by the following two classes of ad-hoc queries: (1) Top-k active events: *"What are the current k most active patterns?"* and (2) Targeted trends: *"Tell me emerging patterns pertaining to specified keywords."*

System comparison. We compare Percolator with Arabesque [7], a state-of-the-art parallel graph mining system. Arabesque does not support mining over dynamic graphs. Thus we develop a "batch" version that interleaves buffered updates and from scratch mining.

Scenario. We invite users to experience the following scenarios. *Performance of* Percolator. Users are invited to configure Percolator and compare the performance of Percolator and Arabesque. We show that Percolator scales well. Over **MAG**, its performance is improved by 2.5 times when the number of workers varies from 2 to 8. Percolator is quite efficient: it takes 245 seconds to process 10 million updates per batch with 8 workers in parallel. In contrast, Arabesque does not run to completion using the same setting.

[1]https://www.elastic.co/products/elasticsearch

[2]available at https://github.com/streaming-graphs/NOUS

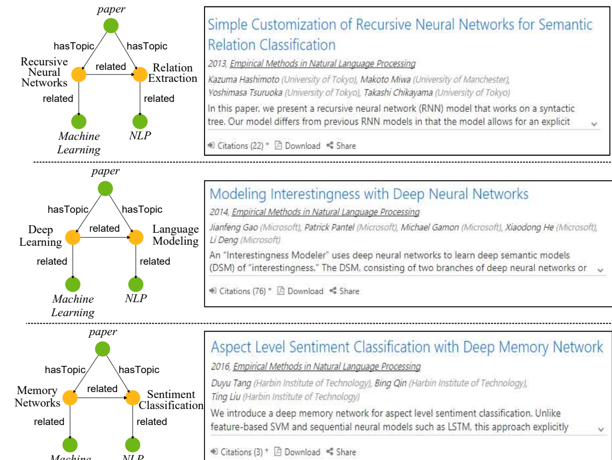

Figure 5: Trend of specified research topics (2013-2016)

Top Event Analysis. Using **Twitter**, we show that given keywords, Percolator effectively tracks top events and their evolution.

Example 2: A top-1 query with three keywords "twitter", "MH17" and "English" finds the most active patterns in English tweets related to "MH17". As shown by the two patterns at bottom in Figure 4, Percolator identifies a shift of twitter topics before and after the event of MH17 plane crash in English tweets. When changing keywords "English" to "Russian", it finds that Russian tweets are less fazed by the event (shown by the two active patterns at the top).

Trend analysis. We next demonstrate that Percolator detects the trend of specified topics, with trend queries over **MAG**.

Example 3: Figure 5 illustrates a trend discovered by Percolator when user specifies trend queries with keywords "Machine Learning", "paper" and "NLP". As verified by the matched triples in **MAG**, the trend shows the most active and relevant research topics changes from "Recursive Neural Network" (2013-14) to "Deep learning" (2014-15) and "Memory networks" (2015-16).

REFERENCES

[1] Björn Bringmann and Siegfried Nijssen. 2008. What is frequent in a single graph?. In *PAKDD*. 858–863.

[2] Yun Chi, Haixun Wang, Philip S. Yu, and Richard R. Muntz. [n. d.]. Moment: Maintaining Closed Frequent Itemsets over a Stream Sliding Window *(ICDM '04)*.

[3] Sutanay Choudhury, Khushbu Agarwal, Sumit Purohit, Baichuan Zhang, Meg Pirrung, Will Smith, and Mathew Thomas. 2017. Nous: Construction and querying of dynamic knowledge graphs. In *ICDE*.

[4] Sutanay Choudhury, Lawrence Holder, George Chin, Khushbu Agarwal, and John Feo. 2015. A selectivity based approach to continuous pattern detection in streaming graphs. *EDBT* (2015).

[5] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. 2014. GRAMI: Frequent Subgraph and Pattern Mining in a Single Large Graph. *PVLDB* (2014).

[6] Wenfei Fan, Jianzhong Li, Jizhou Luo, Zijing Tan, Xin Wang, and Yinghui Wu. 2011. Incremental graph pattern matching *(SIGMOD '11)*.

[7] Carlos HC et al. Teixeira. 2015. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles.*

[8] Xifeng Yan and Jiawei Han. 2002. gSpan: Graph-Based Substructure Pattern Mining. In *ICDM.*