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.

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 rediscovering patterns from scratch upon receiving changes. Given discovered patterns $\Sigma$ over a graph, and a batch of edge updates, it incrementally updates $\Sigma$ 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.

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

KEYWORDS
Graph mining; parallel system; data stream

ACM Reference Format:
We start with the graph streams and the pattern model in Percolator word nodes, if $P$ pertains to $E$. The minimum image support $u$ pertains to a set of user-specified keywords $P$ that the activeness of threshold and $\Delta$ super-pattern $P$ in $\Gamma$. Hence, to support keyword-based pattern discovery, a visualization component is usually tend to be trivial ones. A pattern 

Maximal patterns

Figure 1: Emerging event pattern from News data as triples

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 $\Gamma_T$, a set of keywords $K$, a set of active patterns $\Sigma$ pertain to $K$, (obtained by “from-scratch” discovery), Percolator updates $\Sigma$ in response to a set of edge transactions $\Delta E$ applied to $G$, without rediscovers $\Sigma$ from scratch, and outputs updated $\Sigma$ 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 $\Gamma_T$ is updated, it performs necessary computation that suffices to update $\Sigma$ (Section 3.1). (2) To scale the process over large $\Gamma_T$, it performs incremental mining in parallel, and aggregates the changes to update $\Sigma$ (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 $\Sigma$. 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 $\Gamma_T$ 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 $\Delta E$ 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 $T$ 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 $\Sigma$. (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 $T$. That is, it simulates the exploration of larger patterns in $T$ as $P$ remains to be active.
**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 $\mathcal{T}$ 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 $\Delta E_i$ is fragmented as $\{\Delta E_{i1}, \ldots, \Delta E_{in}\}$, where each $\Delta E_{ij}$ changes $F_i$ to $F_i \cup \Delta E_{ij}$, respectively. (3) The pattern lattice $\mathcal{T}$ is synchronized among all the workers.

**Parallel mining.** Given $\mathcal{T}$ and a batch of updates $\Delta E$, Percolator "parallelizes" the sequential incremental mining (Section 3.1) following a Bulk Synchronous Parallel model, and runs in supersteps. (1) Upon receiving $\Delta E_{ij}$, 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 $\Sigma$ when necessary. It then broadcasts $\Delta \mathcal{P}$ 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 $\Sigma$ 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.
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 Elasticsearch1 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 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.

REFERENCES

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

1https://www.elastic.co/products/elasticsearch
2available at https://github.com/streaming-graphs/NOUS