Percolator: Scalable Pattern Discovery in Dynamic Graphs

Sutanay Choudhury
Pacific Northwest National Laboratory, USA
sutanay@gmail.com

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 Percolator effective supports event and trend analysis in social media streams and research publication analysis, 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). Moreover, pattern discovery should support keyword input, *i.e.,* to discover and track patterns that "cover" 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 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 has the following unique new features that differ from conventional systems.

(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 their matches. This avoids unnecessary computation from scratch, with a cost only determined by these patterns and data changes, independent of the size of graphs.

(3) Scale-up. Percolator is a parallel system implemented on top of Apache Spark. It parallelizes the incremental graph mining in (2) 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.

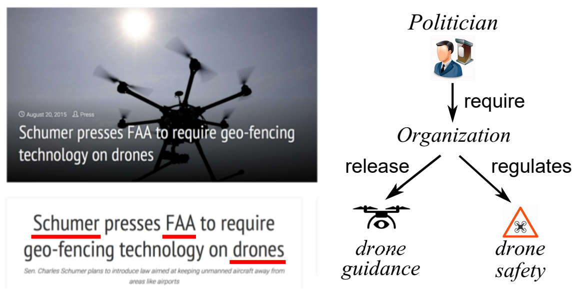
(4) Easy-to-use. The Percolator system is easy to use. It provides a user-friendly GUI, a built-in natural-language query constructor to support keyword-based pattern discovery, a visualization components to inspect and interpret the results.

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 data) 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 STREAMS MODELS

We start with the graph streams and 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 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}.


Schumer presses FAA to require geo-fencing technology on drones

Sen. Charles Schumer plans to introduce law aimed at keeping unmanned aircraft away from areas like airports

Figure 1: Emerging event pattern from News data as triples

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 $\mathcal{K}=\{k_1, \dots k_n\}$, a pattern P pertains to $\mathcal{K}$ if for every keyword k_i $i \in [1, n]$, there exists a keyword node u in P such that $L(u)=k_i$.

Activeness. By default, Percolator extends *minimum image support* [1] to characterize active patterns. 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 (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 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, Act$(P, G_i) \leq$ 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 $\mathcal{K}$ (which can be obtained by "from-scratch" discovery), Percolator updates Σ in response to a set of edge transactions ΔE applied to G, without re-discovery 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 suffice to update Σ. (2) To scale the process over large $\mathcal{G}_T$, it distributes updates and perform incremental mining in parallel, and aggregates the changes to update Σ. We introduce incremental mining in Section 3.1, and its parallel mining in Section 3.2.

3.1 Incremental Mining

Upon receiving updates ΔE, Percolator dynamically identifies a set of "affected" patterns that must be inspected in order to update Σ, and only verifies these patterns. Percolator implements this principle with 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 and process 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 *load shedding* 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.

Pattern lattice. The manager also maintains a pattern lattice $\mathcal{T}$, commonly used in constrained graph mining. The active events Σ are bookkept in $\mathcal{T}$. In addition, it tracks the activeness of the patterns.

Affected Pattern Detector. 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" pattern set $\mathcal{P}$ 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 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.

Upward propagation*:* If Act$(P, G_i) < \theta$ (*i.e.,* becomes inactive due to *e.g.,* edge deletions in ΔE_B), 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 $P \in \mathcal{P}$. (1) If P is hit by edge insertions, it updates 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) If P is hit by edge deletions, it simply checks if the matches of P remains to be valid. To this end, it computes isomorphism from P to a subgraph induced by d-hop (with d the diameter of P) of the deleted

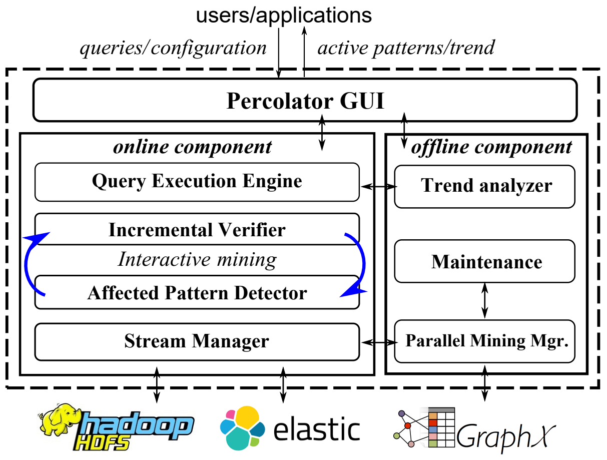


Figure 2: Architecture of Percolator

edges in G_i, without reevaluating P against G_i. (3) Otherwise, P is obtained by either forward or backward propagation, and 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 edge insertions. 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.

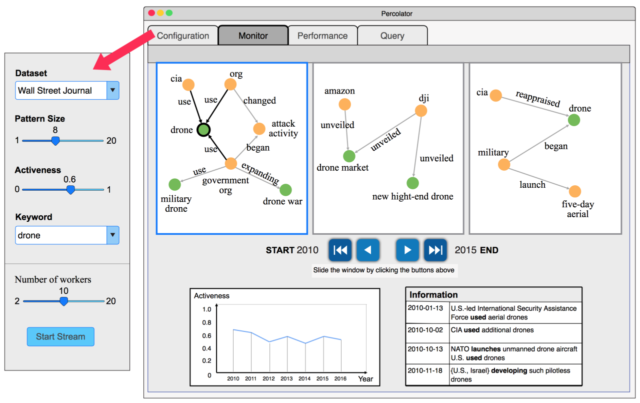
Upon each batch of updates, Percolator interleaves Affected Pattern Detector and Incremental Verifier to "lazily" perform necessary amount of verification. The interaction repeat until no pattern can be added to $\mathcal{P}$. It then reports updated Σ upon request.

Performance. Percolator correctly updates Σ: (1) at any time, it suffices to verify the patterns in $\mathcal{P}$ to update Σ; (2) the Incremental Verifier correctly updates the pattern activeness. Moreover, $\mathcal{P}$ lazily includes the neighbors of affected patterns instead of regenerating all patterns, reducing redundant verifications; and the re-verification of $\mathcal{P}$ visits up to bounded hop of neighbors of ΔE instead of the entire G_i. These ensure the feasiblity of Percolator over large $\mathcal{G}_T$.

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.



Dataset: Wall Street Journal

Pattern Size: 1 — 8 — 20

Activeness: 0 — 0.6 — 1

Keyword: drone

Number of workers: 2 — 10 — 20

Information

2010-01-13	U.S.-led International Security Assistance Force **used** aerial drones
2010-10-02	CIA **used** additional drones
2010-10-13	NATO **launches** unmanned drone aircraft U.S. **used** drones
2010-11-18	{U.S., Israel} **developing** such pilotless drones

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 compute their local activeness, in parallel. For each pattern P that cannot be verified locally, it extends F_j with d-hop neighbors of the "border" nodes of F_j in G and perform local verification, with d the diameter of P.

(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 update Σ when necessary. It then broadcasts $\mathcal{P}$ to all workers.

The above two steps repeats 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.

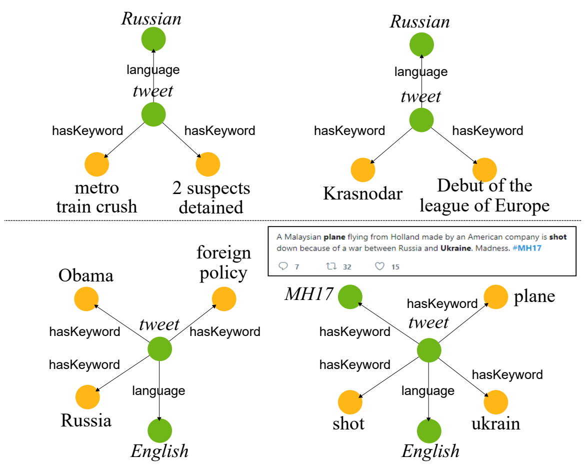
4 SYSTEM OVERVIEW

System architecture. As shown in Figure 2, Percolator consists of three components below.

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.

Percolator *GUI*. The user-friendly Percolator GUI is illustrated in Figure 3. The *configuration panel* receives mining configurations (*e.g.,* activeness threshold, the number of workers). Users can browse and inspect active patterns in the *monitor panel*, which includes the activeness curve, and the details of the triple matches of the selected patterns. The *performance panel* (not shown) visualizes delay time, memory cost and scalability results. Finally, a built-in *Query panel* allows users to issue natural-language style analytical queries, supported by built-in query parsers.



A Malaysian **plane** flying from Holland made by an American company is **shot** down because of a war between Russia and **Ukraine**. Madness. #MH17

7 32 15

Figure 4: Top-k Twitter Patterns

Implementation. Percolator is implemented in Scala, and built on top of Apache Spark and HDFS. Percolator uses core functions in GraphX library to implement incremental mining. The graph stream is represented as distributed arrays (RDDs) managed by Spark. To support fast stream access, Percolator uses Elasticsearch[1], an in-memory database to manage the intermediate results (*e.g.,* the mapping in Stream Manager) as key-value pairs. In addition, Percolator connects to our prior work of graph construction [3] and processing [4] as input and query interfaces, respectively.

The system Percolator is deployed on a cluster of 16 nodes (with one serving as the coordinator), each equipped with an Intel Xeon processor (2.3 GHz) with 16 cores and 64 GB memory.

5 DEMONSTRATION OVERVIEW

We next present our demonstration plan. The target audience of the demo includes anyone who is interested in understanding complex event and trend over data streams.

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.

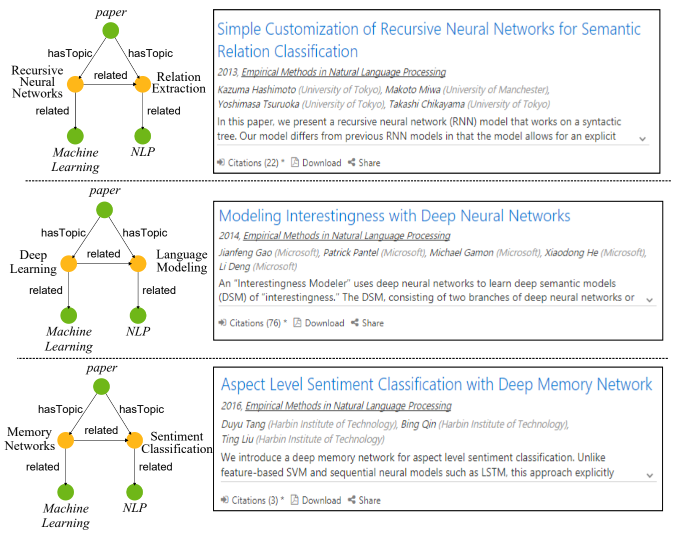
Ad-hoc queries. We invite users to inspect patterns discovered by the following two classes of ad-hoc queries . (1) Top-k active events: *"what are the current k most active patterns?"* and (2) Targeted trends: *"tell me emerging patterns pertaining to specified keywords."*

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

Scenario. We invite users to experience the following scenarios.

Performance of Percolator. We demonstrate the efficiency and scalability of Percolator, and the impact of key factors (*e.g.,* pattern size, activeness threshold, number of workers). Users are invited to configure Percolator and compare the performance of Percolator and Arabesque. We show that Percolator scales well. For example, over

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



Simple Customization of Recursive Neural Networks for Semantic Relation Classification

2013, Empirical Methods in Natural Language Processing

Kazuma Hashimoto (University of Tokyo), Makoto Miwa (University of Manchester), Yoshimasa Tsuruoka (University of Tokyo), Takashi Chikayama (University of Tokyo)

In this paper, we present a recursive neural network (RNN) model that works on a syntactic tree. Our model differs from previous RNN models in that the model allows for an explicit

Citations (22) * Download Share

Modeling Interestingness with Deep Neural Networks

2014, Empirical Methods in Natural Language Processing

Jianfeng Gao (Microsoft), Patrick Pantel (Microsoft), Michael Gamon (Microsoft), Xiaodong He (Microsoft), Li Deng (Microsoft)

An "Interestingness Modeler" uses deep neural networks to learn deep semantic models (DSM) of "interestingness." The DSM, consisting of two branches of deep neural networks or

Citations (76) * Download Share

Aspect Level Sentiment Classification with Deep Memory Network

2016, Empirical Methods in Natural Language Processing

Duyu Tang (Harbin Institute of Technology), Bing Qin (Harbin Institute of Technology), Ting Liu (Harbin Institute of Technology)

We introduce a deep memory network for aspect level sentiment classification. Unlike feature-based SVM and sequential neural models such as LSTM, this approach explicitly

Citations (3) * Download Share

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

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 complete over a small fraction of **MAG** using the same setting.

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 most active patterns in English twitters 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 twitters. 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] B. Bringmann and S. Nijssen. What is frequent in a single graph? In *PAKDD*, pages 858–863, 2008.
[2] Y. Chi, H. Wang, P. S. Yu, and R. R. Muntz. Moment: Maintaining closed frequent itemsets over a stream sliding window. ICDM '04.
[3] S. Choudhury, K. Agarwal, S. Purohit, B. Zhang, M. Pirrung, W. Smith, and M. Thomas. Nous: Construction and querying of dynamic knowledge graphs. In *ICDE*, 2017.
[4] S. Choudhury, L. Holder, G. Chin, K. Agarwal, and J. Feo. A selectivity based approach to continuous pattern detection in streaming graphs. *EDBT*, 2015.
[5] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis. GRAMI: frequent subgraph and pattern mining in a single large graph. *PVLDB*, 2014.
[6] W. Fan, J. Li, J. Luo, Z. Tan, X. Wang, and Y. Wu. Incremental graph pattern matching. SIGMOD '11, 2011.
[7] C. H. e. a. Teixeira. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.
[8] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *ICDM*, 2002.