Dynamic Interaction Graphs with Probabilistic Edge Decay

Wenlei Xie[1], Yuanyuan Tian[2], Yannis Sismanis[3], Andrey Balmin[4], Peter J. Haas[2]
[1]Cornell University
wenleix@cs.cornell.edu
[2]IBM Almaden Research Center
{ytian, phaas}@us.ibm.com
[3]Google, Inc.
yannis@google.com
[4]Platfora, Inc.
andrey@platfora.com

***Abstract*—A large scale network of social interactions, such as mentions in Twitter, can often be modeled as a "dynamic interaction graph" in which new interactions (edges) are continually added over time. Existing systems for extracting timely insights from such graphs are based on either a cumulative "snapshot" model or a "sliding window" model. The former model does not sufficiently emphasize recent interactions. The latter model abruptly forgets past interactions, leading to discontinuities in which, e.g., the graph analysis completely ignores historically important influencers who have temporarily gone dormant. We introduce TIDE, a distributed system for analyzing dynamic graphs that employs a new "probabilistic edge decay" (PED) model. In this model, the graph analysis algorithm of interest is applied at each time step to one or more graphs obtained as samples from the current "snapshot" graph that comprises all interactions that have occurred so far. The probability that a given edge of the snapshot graph is included in a sample decays over time according to a user specified decay function. The PED model allows controlled trade-offs between recency and continuity, and allows existing analysis algorithms for static graphs to be applied to dynamic graphs essentially without change. For the important class of exponential decay functions, we provide efficient methods that leverage past samples to incrementally generate new samples as time advances. We also exploit the large degree of overlap between samples to reduce memory consumption from $O(N)$ to $O(\log N)$ when maintaining N sample graphs. Finally, we provide bulk-execution methods for applying graph algorithms to multiple sample graphs simultaneously without requiring any changes to existing graph-processing APIs. Experiments on a real Twitter dataset demonstrate the effectiveness and efficiency of our TIDE prototype, which is built on top of the Spark distributed computing framework.**

I. Introduction

In a world of booming social networking services and pervasive mobile devices, electronic records of social interactions between people are being generated at ever-increasing rates. A network of social interactions often can be modeled as a "dynamic interaction graph" in which new interactions, represented by edges, are continually added. Examples include phone-call graphs generated by telecommunication service providers, message graphs from social networking sites, and mention-activity graphs formed by Twitter users mentioning one another in their tweets. Dynamic interaction graphs are very different from traditional social graphs, such as the friendship graphs from social networking sites, in which the social relationships evolve gradually. Real-life social interactions, such as phone calls and tweets, happen much more rapidly. For example, as of January 2014, 58 million tweets were generated daily on Twitter. In essence, a dynamic interaction graph can be viewed as a data stream of interactions.

Enterprises are analyzing streams of interactions for insights relevant to real-time decision making. This poses a significant challenge to algorithm design, because the overwhelming majority of graph algorithms assume static graph structures. As a result, most existing systems designed for graph stream analysis [1], [2], [3] successively process a sequence of static views, or "snapshots", of a dynamic graph, where a snapshot comprises all interactions seen so far. As time advances, the result is updated incrementally, if possible, or else by re-running the algorithm from scratch. We call this simple model the "snapshot model".

A key drawback of this approach is the ever-increasing size of the snapshots. Graph analysis is usually much more complex than maintenance of simple aggregates over a stream of data, and the memory usage of virtually all available graph algorithms increases with increasing graph size. As a result, computation and memory resources quickly run out as interactions are added to the dynamic graph. Another drawback of the snapshot model is the *recency* problem: as time progresses, the proportion of stale data in the snapshot becomes ever larger and analysis results increasingly reflect out-of-date characteristics of the dynamic graph.

One simple approach to reducing the size of the snapshots and enforcing recency requirements is to use a "sliding-window" model, where only recent interactions that happen within a small fixed-size time window are considered in the analysis. This simplistic cut-off approach completely forgets historical interactions and thus loses the *continuity* of the analytic results with time. Historical interactions may be less relevant to today's decision making, but do not completely lack value, especially in the aggregate. The following example demonstrates the drawbacks of the snapshot and sliding-window models.

Example 1 (Influence Analysis)**.** *An advertising company is analyzing the mention-activity graph from Twitter to identify key influencers with respect to skiing equipment. A key influencer is someone who has many interactions with other users on skiing-related topics. Consider the following three users:*

- Alice joined Twitter five years ago and has been regularly and frequently interacting with other users on skiing-related topics since then. She has been inactive for the last three weeks because she is on a skiing trip in Europe.

- Bob joined Twitter two months ago, and since then has had many interactions on skiing related topics.

- Carol also joined Twitter five years ago. She was extremely active on skiing topics for the first six months, but then lost interest and has never tweeted about skiing again.

Figure 1 illustrates the frequency of interactions for the

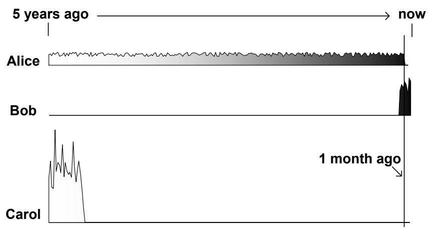


Fig. 1. Influence analysis example

three users over time. Intuitively, Alice is a steady influencer who is temporarily dormant. Bob can be viewed as a rising star. Although it is unknown whether Bob can maintain his influence in the future (instead of becoming another Carol), as of now, he should be considered a target for ads. Obviously, Carol is not an influencer at present.

Under the snapshot model, there is no distinction between recent interactions and old ones. Alice, correctly, is an influencer. Bob, incorrectly, is not an influencer because the number of his interactions is relatively small compared to the cumulative interaction counts of the old-timers. On the other hand, the fact that Carol has had a huge number of interactions incorrectly makes her an influencer, even though all of these interactions are in the remote past (but perhaps she should receive an ad just in case). By comparison, if we use a sliding-window model with a one-month window length, Bob will be an influencer, but Alice will not be considered as an influencer at all. Carol will never receive an ad.

To address the above issue, we take an approach inspired by the literature on sampling from data streams. Specifically, we consider *temporally biased sampling*, as was proposed for ordinary (non-graph) data streams in [4]. The general idea is to sample data items according to a probability that decreases over time, so that the sample contains a relatively high proportion of recent data points. In our example, the gray levels in Figure 1 illustrate temporally biased sampling rates (darker shades correspond to higher inclusion probabilities). Carol's historical interactions will be significantly downgraded in the influence analysis (but not completely ignored). Bob's recent interactions will be valued more. Although Alice is not active right now, her consistent interactions throughout time help her maintain influence.

Temporally biased sampling is especially appealing for analyzing dynamic interaction graphs. First, sampling deals gracefully with the increasing size of a dynamic graph. Second, temporal biasing creates samples with more recent interactions (recency) yet still keeps some old interactions to provide the necessary context for the analysis (continuity). Finally, users can apply any existing algorithm for static graphs as-is, avoiding the need to design new, even more complex algorithms that attempt to satisfy recency and continuity requirements.

Although the idea of temporally biased sampling is not new, we are the first to apply it to the analysis of dynamic graphs. In particular, as discussed in what follows, we refine the generic temporally biased sampling approach by exploiting graph-specific properties—especially the overlapping of edges between graphs—to achieve space and time efficiencies. We also describe challenges and solutions when building a distributed system to support this important new functionality.

We formalize temporally biased sampling for dynamic graphs via a *probabilistic edge decay* (PED) model. Under this model, we sample interactions from the current snapshot. Each interaction has an independent probability of appearing in the resulting sample graph, and this probability is non-increasing with the age of the interaction. The PED model subsumes both snapshot and sliding-window models; see Section III. To control sampling variability, we generate multiple independent sample graphs, execute the analytic algorithm on each one, and then average (or otherwise aggregate) the results. The practical advantages of this approach can be significant. In an empirical study on a real Twitter dataset (see Section VII-B1 below), we found that a significant fraction of the top influencers found via the PED approach were either steady but temporarily dormant influencers like Alice or rising stars like Bob; these important sets of influencers would be overlooked under a snapshot or sliding-window approach, respectively.

We have developed an end-to-end system for dynamic graph analysis, called TIDE, that embodies the above ideas. TIDE is implemented on top of the Spark distributed processing system [5], leveraging its native streaming [6] and graph processing [7] support. TIDE allows users to analyze dynamic graphs using existing algorithms for static graphs; moreover, analyses can be specified using existing APIs for batch graph processing systems. Empirical studies on a real Twitter dataset demonstrate the effectiveness and efficiency of the system. TIDE is the first distributed system to systematically support probabilistic edge decay for analyzing dynamic graphs.

The contributions of this paper are as follows:

- We formalize a general PED model for dynamic graphs that implements temporally biased sampling and subsumes existing models.
- We develop incremental sample-maintenance methods for PED models with exponential decay functions.
- We exploit overlap between sample graphs to store the sample set in a space-efficient manner.
- We provide a bulk graph execution model to efficiently analyze multiple samples of dynamic interaction graphs simultaneously.
- We exploit overlap between the realizations of a sample graph at successive time points to allow efficient incremental graph analysis.
- We show how to efficiently implement the TIDE system using Spark (with some modifications).
- We provide experiments on real-world data to assess our new techniques.

II. Dynamic Interaction Graphs: Existing Models

In this section we formalize both snapshot and sliding window models for dynamic interaction graphs. Given a time domain T, a *dynamic interaction graph* (or *dynamic graph* for short) is defined as $G = (V, E)$, where V is a set of vertices and $E \subseteq V \times V \times T$ is a set of time-stamped edges. The presence of an edge $e = (u, v, t) \in E$ indicates that vertex u interacts with vertex v at time t. We denote by $t(e)$ the time stamp associated with edge e. Note that there can be multiple edges from u to v but with different timestamps. In addition, there may be other attributes associated with the vertices and edges of G.

In Twitter, for example, Alice *mentions* Bob in a tweet if the tweet includes the string "@Bob", and this *mention interaction* indicates a certain level of attention paid by Alice to Bob [8]. Such mention interactions in Twitter can be modeled as a dynamic graph. The vertices are Twitter users, and an edge from Alice to Bob with timestamp t means Alice mentioned Bob in a tweet at time t. The actual tweet can be modeled as an attribute associated with this edge, and user profiles for Alice and Bob can be captured as vertex attributes. Note that arriving edges sometimes introduce new vertices into a dynamic graph; for simplicity, we consider such vertices to already exist in the dynamic graph, but with no prior adjacent edges.

A *snapshot* of a dynamic graph G at time t is defined as $G_t = (V, E_t)$, where $E_t = \{e \mid e \in E \wedge t(e) \leq t\}$. Similarly, a *window* of G from time t to t' is defined as $G_{t,t'} = (V, E_{t,t'})$, where $E_{t,t'} = \{e \mid e \in E \wedge t \leq t(e) \leq t'\}$. In the *snapshot model*, an analytic function F applied to a dynamic graph G at time t is actually applied to the snapshot G_t, with the result $F(G_t)$. As time advances to t', the result is updated to $F(G_{t'})$ either by computing it from scratch on $G_{t'}$ or by incrementally updating the result from $F(G_t)$ to $F(G_{t'})$. In the *sliding-window model*, the function F is applied to $G_{t-w,t}$, where w is a fixed *window size*, i.e., the analysis only considers interactions that happened within the last w time units.

Observe that both models embody a binary view of an edge's role in an analysis; it is either included for analysis or not. An included edge has the same importance as any other edge, regardless of how outdated it is. As mentioned previously, this simplistic view makes it impossible to satisfy both recency and continuity requirements simultaneously. In contrast, temporally biased sampling of the dynamic graph provides a probabilistic view of an edge's role: edges from past to present all have chance to be considered (continuity) but outdated edges are less likely to be used (recency) in an analysis, so that the influence of an edge decays over time. In the following section, we describe the probabilistic edge decay (PED) model for temporally biased sampling.

III. The PED Model

When applying a function to a dynamic graph at time t under the PED model, an edge e with a timestamp $t(e) \leq t$ has an independent probability $P^f(e)$ of being included in the analysis, where $P^f(e) = f\big(t - t(e)\big)$ for a non-increasing *decay function* $f : \Re_+ \mapsto [0, 1]$. As time advances, e's age $t - t(e)$ increases and the inclusion probability $P^f(e)$ either decreases or remains unchanged. Note that the snapshot model and the sliding-window model are two special cases of the PED model with $f \equiv 1$ and $f(a) = I(a \leq w)$ respectively, where $I(X)$ denotes the indicator of event X. In general, we can require that f be positive and strictly decreasing. Then, at any time t, every edge e with $t(e) \leq t$ has a non-zero chance of being included in the analysis (continuity) but an edge becomes increasingly unimportant in the analysis over time, so that newer edges are more likely to participate in the analysis (recency).

Formally, let $G = (V, E)$ be a dynamic graph and f a decay function. For $t \geq 0$, denote by $\mathcal{G}_t = \{\, (V, E') : E' \subseteq E_t \,\}$ the set of $2^{|E_t|}$ *possible graphs* at time t. (Here E_t is defined as in Section II and $|E_t|$ denotes the number of edges in E_t; we suppress the underlying dynamic graph G in the notation.) Define the *possible-graph distribution* $\mathbb{P}_{f,t}$ over $\mathcal{G}_t$ by setting

$$\mathbb{P}_{f,t}(G') = \prod_{e \in E'} f\big(t - t(e)\big) \prod_{e \in E_t - E'} \big[1 - f\big(t - t(e)\big)\big] \quad (1)$$

for $G' = (V, E') \in \mathcal{G}_t$. A *sample graph* at time t (with respect to f) is defined as a graph drawn from the distribution $\mathbb{P}_{f,t}$. In the *PED model*, an analytic function F applied to G at time t is actually applied to $N \geq 1$ independent and identically distributed (i.i.d.) sample graphs $G_t^{f,1}, G_t^{f,2}, \ldots, G_t^{f,N}$ to yield i.i.d. results $F(G_t^{f,1}), F(G_t^{f,2}), \ldots, F(G_t^{f,N})$. These results can be used to control the variability introduced by the sampling process. In the simplest cases, the results can be averaged together. For example, if F returns the influence score for each person in an interaction graph, then one might want to compute the average per-person influence score at time t. In general, analysts can decide whether and how they want to aggregate the results into one result; see Section VII for further discussion.

In what follows, we focus on the important class of *exponential* decay functions of the form $f(a) = p^a$ for some $0 < p < 1$. We call p the *decay probability*. In general, the exponential decay of edges captures most application scenarios and has been widely adopted in practice [9], [10], [11]. Moreover, exponential edge decay guarantees that the space requirement for storing the dynamic graph is bounded with high probability; see Section IV-B.

For simplicity, we adopt a discretized time approach that has been widely used in existing work [6], [12]. Specifically, the continuous time domain is partitioned into intervals of length Δ, and the dynamic graph is observed only at times $\{\, k\Delta : k \in \mathcal{N} \,\}$, where $\mathcal{N} = \{\, 0, 1, 2, ... \,\}$. Moreover, all edges that arrive in an interval $\big[k\Delta, (k+1)\Delta\big)$ are treated as if they arrived at time $k\Delta$, i.e., at the start of the interval. Thus we can take $T = \mathcal{N}$ for the time domain, $k \in \mathcal{N}$ to represent the age of an edge, and $f(k) = p^k$ to represent the exponential decay function. Moreover, updates to a dynamic graph can be viewed as arriving in a stream of batches $B_0, B_1, B_2, \ldots$, where all incoming edges in batch B_i have time stamp i.

IV. Maintaining Sample Graphs

In this section we describe how to efficiently maintain the set of N sample graphs over time. The key ideas are to incrementally update the sample graphs and to exploit overlaps between the sample graphs at a given time point by storing the graphs in an aggregated form. We first describe our general approach to incremental maintenance of the sample graphs and then describe how these graphs are stored in a space-efficient "aggregate graph". We then combine these techniques to obtain specific algorithms for eager and lazy updating of the set of sample graphs.

A. Incremental Updating: General Approach

As time advances from t to $t + 1$, a naive way to update the results is to materialize N independent sample graphs from scratch and then analyze them. However, generating N samples from the ever larger snapshot graph is prohibitively expensive. An incremental approach for computing sample

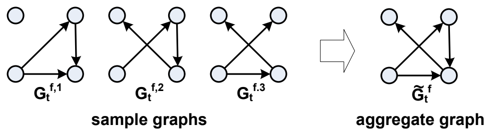


Fig. 2. Example aggregate graph

graphs rests on the following theorem, the proof of which is straightforward.

Theorem 1. *For $f(k) = p^k$, let $G_t^{f,i} = (V, E_t^{f,i})$ be the ith sample graph at time t, so that $G_t^{f,i}$ has probability distribution $\mathbb{P}_{f,t}$ given in* (1)*, and let B_{t+1} be the batch of incoming edges at time $t+1$. Let $G' = (V, S_p(E_t^{f,i}) \cup B_{t+1})$, where $S_p(E_t^{f,i})$ is a Bernoulli sample of $E_t^{f,i}$ with sampling probability p. Then G' has distribution $\mathbb{P}_{f,t+1}$, that is, G' can be viewed as a sample graph at time $t+1$.*

This result provides an efficient way of constructing $G_{t+1}^{f,i}$ from $G_t^{f,i}$. Instead of generating $G_{t+1}^{f,i}$ from scratch, we only need to subsample the edge set of $G_t^{f,i}$ and combine the subsample with the edges in the arriving batch B_{t+1}. It follows immediately from the theorem that the N sample graphs at $t+1$ generated by this incremental updating scheme are the desired N independent sample graphs, provided that the N sample graphs at t are independent and each subsampling process is executed independently. The instantiations of the ith sample graph at times t and $t+1$ overlap significantly. Indeed, it is not hard to see that, in expectation, $G_t^{f,i}$ shares a fraction p of its edges with $G_{t+1}^{f,i}$ under incremental updating.

B. The Aggregate Graph

Besides the overlap between instantiations of a sample graph at two consecutive time points, there is also overlap between different sample graphs at the same time point. Suppose, for example, that each update batch is of size M. Denote by $S_{p,M,t}$ the number of edges in a sample graph at time t with decay function $f(k) = p^k$, and assume throughout that t is large. We then have $E[S_{p,M,t}] \approx M \sum_{k=0}^{\infty} p^k = \frac{M}{1-p}$. Moreover, $S_{p,M,t}$ has a Poisson-Binomial distribution, so that, specializing the high-accuracy "refined normal approximation" in [13], we have for large t and $j = 0, 1, \ldots$ that $P(S_{p,M,t} \leq j) \approx \Phi(y) + \gamma(1-y^2)\phi(y)$, where Φ and ϕ are the cumulative distribution function and probability density function of a standard (mean 0, variance 1) normal distribution, $y = (j+0.5-\mu)/\sigma$, $\gamma = (\mu/\sigma^3)(p^3-p^2+p)/(1+p-p^3-p^4)$, $\mu = M/(1-p)$, and $\sigma^2 = Mp/(1-p^2)$. For the moderate values of p and large values of M encountered in practice—e.g., $p = 0.8$ and $M = 13.9$ million in our experiments—the distribution of $S_{p,M,t}$ is sharply concentrated around its mean. With the above values of p and M, for example, $S_{p,M,t}$ lies within roughly $\pm 1\%$ of its mean with a probability exceeding 99.99%. Denoting by $S'_{p,M,t}$ the number of edges shared by one sample graph with another, we have $E[S'_{p,M,t}] \approx M \sum_{k=0}^{\infty} p^{2k} = \frac{M}{1-p^2}$, because an edge with age k has a probability $p^k \cdot p^k = p^{2k}$ of appearing in both sample graphs at the same time. Again, there is sharp concentration about the mean, and so the expected fraction of shared edges is $E[S'_{p,M,t}/S_{p,M,t} \mid S_{p,M,t} > 0] \approx \frac{M}{1-p^2} / \frac{M}{1-p} = \frac{1}{1+p} > \frac{1}{2}$, and similarly $S'_{p,M,t}/S_{p,M,t} > 1/2$ with high probability.

Given the significant overlap between different sample graphs at a time point, we see that naively maintaining N sample graphs $G_t^{f,1}, G_t^{f,2}, \ldots, G_t^{f,N}$ separately incurs much redundancy. Instead, we can store the N sample graphs as a single *aggregate graph* $\tilde{G}_t^f = (V, \bigcup_{i=1}^{N} E_t^{f,i})$, where the edge sets of the sample graphs are simply unioned. Figure 2 shows an example aggregate graph comprising three sample graphs. The attributes for an edge that appears in multiple sample graphs need only be stored once in the aggregate graph. For each aggregate edge, we keep track of the sample graph(s) to which the edge belongs.

In contrast to the continually increasing memory requirement in the snapshot model, the PED model has a bounded memory requirement as new edges are added over time, provided that the update batch at each time stamp is bounded. Denoting by M the maximum size of an update batch, we see from our earlier analysis that the size of each sample graph is bounded by $\frac{M}{1-p}$ with very high probability. It follows that even the naive approach of storing N sample graphs separately has a sharp probabilistic upper bound of $\frac{MN}{1-p}$ edges.

To analyze the expected space requirement for the aggregate graph, first observe that, under incremental updating, an edge e that does not appear in the aggregate graph at time t will not appear in the aggregate graph for $t' > t$. As a result, we can establish a memory bound that is significantly smaller than that of the naive approach.

Theorem 2. *Let M be the maximum size of an update batch, and $f(k) = p^k$ be an exponential decay function. Then the expected number of edges in the aggregate graph of N sample graphs at any time is bounded by $M\lceil \log_{\frac{1}{p}}(N) \rceil + \frac{M}{1-p}$.*

Proof: Based on the definition of the exponential decay function, an edge whose age is k just prior to a given update of a sample graph will be removed from the graph with probability $1-p^k$. Thus the edge has probability $1-(1-p^k)^N$ of appearing in at least one of N sample graphs after an update. The expected total number of edges in the aggregate graph is therefore bounded by $\sum_{k=0}^{\infty} M\big(1-(1-p^k)^N\big)$. Setting $K = \lceil \log_p \frac{1}{N} \rceil = \lceil \log_{\frac{1}{p}} N \rceil$, we have

$$\begin{aligned}
&\sum_{k=0}^{\infty} M(1-(1-p^k)^N) \\
=&M \sum_{k=0}^{K-1} 1-(1-p^k)^N + M \sum_{k=K}^{\infty} 1-(1-p^k)^N \\
\leq& MK + M \sum_{k=K}^{\infty} Np^k = MK + \frac{MNp^K}{1-p} \\
\leq& M\lceil \log_{\frac{1}{p}}(N) \rceil + \frac{M}{1-p},
\end{aligned}$$

where $(1-p^k)^N \geq 1 - Np^k$ by Bernoulli's inequality. ■

The above theorem provides an upper bound (for all time points) on the expected memory consumption when using the aggregate graph to maintain N sample graphs. Observe that the expected number of edges that need to be stored is reduced from $O(MN)$ for the naive approach to $O(M \log N)$. For example, when $p = 0.8$ $N = 96$, and $M = 10$ million, the expected storage requirement would be 4.8 billion edges for

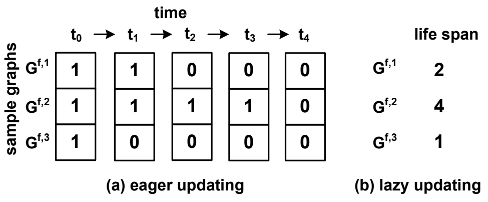


Fig. 3. Incremental updating for one edge

the naive approach but only about 250 million edges using the aggregate graph. Arguments as before show that, typically, the above expected storage complexities for the two approaches also yield high probability upper bounds. In the aggregate graph, we can use a bit array for each edge e to indicate the sample graphs in which e appears; this additional storage is worthwhile because of the savings from not storing redundant edges and their attributes. In fact, as we show in Section IV-D below, we can even avoid storing the bit array for an edge and simply materialize it whenever it is needed.

C. Eager Incremental Updating

We can now combine incremental updating techniques with the aggregate graph to obtain specific algorithms for maintaining a set of N sample graphs. Our first approach is called the *eager* incremental updating method and is a straightforward implementation of the process described in Theorem 1.

We store the N sample graphs together in the aggregate graph, and attach a bit array of size N, denoted as β, to each edge e in the aggregate graph, to indicate the sample graphs to which this edge belongs. Specifically, $e.\beta[i] = 1$ means that e appears in the ith sample graph and $e.\beta[i] = 0$ otherwise. As shown in Figure 3(a), whenever a new edge e is first added to the dynamic graph, $e.\beta[i] = 1$ for all i, because the edge appears in all sample graphs. As time goes by, e gradually disappears from some sample graphs. At each batch arrival time, we apply a Bernoulli trial with probability p on e for each sample graph where e still appears. Thus, at each update, we scan through the bit array and, for each bit that equals 1, we set it to 0 with probability $1 - p$. Once β contains all 0s, we remove the edge from the aggregate graph.

This eager incremental updating method is simple and straightforward, but it requires a bit array of size N for each edge in the aggregate graph. This motivates our second approach, the *lazy* incremental updating method.

D. Lazy Incremental Updating

The lazy incremental updating method avoids materializing the bit arrays based on the observation that the *life span* L_e^i of edge e in the ith sample graph follows a geometric distribution; the life span is the time from when the edge arrives until it is permanently removed from the aggregate graph via a Bernoulli subsampling step. That is, $P(L_e^i = l) = p^{l-1}(1-p)$ for $l \in \{1, 2, \dots\}$. Note that $L_e^i \geq 1$ because e always appears in all of the sample graphs when it first arrives. Figure 3(b) shows the life spans in different sample graphs of the example edge in Figure 3(a).

Based on this observation, we can simplify the incremental updating process. For an edge e that has just been added to the ith sample graph, we directly sample the lifetime L_e^i. Then, based on the edge's time stamp $t(e)$ and the life span L_e^i, we know exactly when it will disappear from the ith sample graph. Observe, however, that we need to keep track of the life span for each edge in each sample graph. A naive approach would use N integers per edge, which is an even worse storage requirement than for the N bits per edge in the eager incremental updating method.

We avoid the storage problem by using a lightweight method to deterministically materialize the N integers whenever they are needed, while maintaining their mutual statistical independence. Specifically, we exploit a 64-bit version of the MurmurHash3 random hash function [14]. Given the unique combination of an edge ID and a sample graph ID, MurmurHash3 can deterministically and efficiently generate a 64-bit integer. Moreover, the integers generated for different (edge ID, graph ID) combinations appear random enough to pass the highly rigorous TestU01 [15] test suite for pseudorandom number generators. We use standard techniques to transform the pseudorandom 64-bit integers produced by MurmurHash3 into pseudorandom samples from the geometric distribution; see, for example, [16, p. 469].

E. General Decay Functions

The foregoing discussion can be generalized to decay functions other than the exponential function $f(k) = p^k$. Indeed, for an arbitrary decay function given by $f(k) = \theta_k$ with $1 = \theta_0 \geq \theta_1 \geq \theta_2 \geq \cdots$, we can use an eager incremental updating scheme as before, but with a Bernoulli sampling rate $p_k = \theta_k/\theta_{k-1}$ when processing batch B_k for $k \geq 1$. If $\sum_k \theta_k < \infty$, then arguments almost identical to those given before show that the number of edges in the aggregate graph is bounded in expectation and with high probability.

V. Bulk Analysis of Sample Graphs

The previous section discussed how to efficiently maintain a set of N sample graphs. In this section, we focus on how to efficiently execute analysis algorithms on these graphs. An important design goal of our system is to provide, for dynamic graphs, the same familiar analytics interfaces used in systems for managing static graphs. We therefore adopt the popular vertex-centric iterative computation model used in static graph processing systems such as Pregel [17], GraphLab [18], Trinity [19] and GRACE [20]. Under this computation model, a user-defined *compute*() function is invoked on each vertex v to change the state of v and of v's adjacent edges; changes to other vertices are propagated through either message passing (e.g., in Pregel) or scheduling of updates (e.g., in GraphLab). This computation is carried out iteratively until there is no status change for any vertex. Given this computation model, we describe techniques both for bulk execution of analytics and for incremental updating of analytical results as time progresses.

A. Bulk Graph Execution Model

The most straightforward way to analyze N sample graphs is to materialize each sample graph from the current aggregate graph and apply the analytic function of interest to each individual sample graph. However, this naive approach ignores the significant overlap between the sample graphs, as discussed in Section IV-A. The key observation is that similar topologies lead to similar vertex and edge states among the different sample graphs during the iterative computation.

To take advantage of the similarities among sample graphs, we propose a bulk execution model on multiple sample graphs. We first partition the N sample graphs into one or more *bulk sets* comprising s ($\leq N$) sample graphs. For each bulk set, we combine the s sample graphs into a *partial aggregate* graph, and process the partial aggregate graph as a whole instead of processing the s sample graphs individually. The state of a vertex or an edge in the partial aggregate graph is an array of the states of the corresponding vertex or edge in the s sample graphs. If an edge does not appear in a sample graph, then the associated array element is *null*.

Algorithm 1: Bulk Graph Execution Model

```
input : A vertex $v$ in a partial aggregate graph of $s$ sample graphs,
        its adjacent edges $E_v$, and its incoming messages inMsgs
1  initialize msgs= $\emptyset$; // each element is in the form
       <dest vertex id, message, sample graph id>
2  for i=1 to $s$ do
3      construct a new vertex $v_i$ where $v_i.state = v.state[i]$;
4      $inMsgs_i$=inMsgs.getMsgsForGraph(i);
5      initialize $v_i$'s adjacent edges $E_{v_i} = \emptyset$;
6      foreach $e \in E_v$ do
7          if $e$ is in the ith sample graph then
8              construct $e_i$ where $e_i.state = e.state[i]$;
9              $E_{v_i}$.add($e_i$);
10     orgMsgs=compute($v_i$, $E_{v_i}$, $inMsgs_i$);// call user
           defined function
11     msgs.add(attachGraphID(orgMsgs, i));
12     $v.state[i] = v_i.state$;
13     foreach $e_i \in E_{v_i}$ do
14         $e$ is the corresponding edge in $E_v$;
15         $e.state[i] = e_i.state$;
output: The combined messages grouped by dest vertex id:
        grpMsgs=msgs.groupByDest()
```

Computation at a vertex v in the partial aggregate graph proceeds by looping through the s sample graphs, reconstructing the set of v's adjacent edges in each sample graph and applying the *compute*() function. The resulting updates to other vertices are then grouped by the destination vertex ID and the combined updates are propagated via message passing or scheduling of updates. Consider, for example, a message passing setting, and suppose that the bulk computation on a vertex v results in two messages to destination vertex u: $\langle u, m_1 \rangle$ for the ith sample graph and $\langle u, m_2 \rangle$ for the jth sample graph. Then a combined message $\langle u, \{(m_1, i), (m_2, j)\} \rangle$ is sent to u. Algorithm 1 demonstrates the bulk execution model when message passing is used for update propagation. After one bulk set is complete, we proceed to the next bulk set until all of the N sample graphs are processed.

The benefit of the bulk graph execution model is multifold. First, extracting and loading sample graphs from the full aggregate graph in groups of size s amortizes the nontrivial overheads of this pre-analysis step. Second, because graph traversal requires many random memory accesses, bulk execution of computations on the same vertex across different sample graphs results in local computations that yield improved caching behavior. Finally, the similar message values in a combined message from one vertex to another create

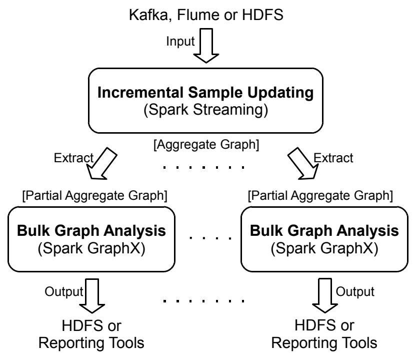


Fig. 4. Overall Data Pipeline of TIDE System

opportunities for compression during communication over the network. Likewise, compression can be applied when persisting similar values in the state array for a vertex or edge on disk for purposes of checkpointing.

The number s of sample graphs in a bulk set is a tunable system parameter that trades off space minimization and computational efficiency. A larger value of s enables shared computation among more sample graphs and hence more benefit from compression of vertex/edge states and combined updates, but also leads to higher memory requirements for each bulk execution. We discuss how to choose s in Section VII.

B. Incremental Graph Analysis

Now that we have an efficient way to analyze the N sample graphs at time t, can we exploit the results at t to more efficiently generate results at $t+1$? We have shown that the instantiations of a given sample graph at two consecutive time points share a large number of common edges, which leads to similar vertex and edge states during the graph computation. For iterative graph algorithms such as Katz centrality [21] and PageRank [22] computation, this provides an opportunity to use the ending vertex and edge states at time t as the starting states for the iterative computation at time $t+1$. These improved starting states can lead to faster convergence. One caveat is that some algorithms do not work correctly under this incremental scheme [23], so that recomputation from scratch is required. Existing dynamic graph processing systems [1], [2], [24] encounter the same issue.

VI. Implementation on Spark

In this section, we provide a brief overview of Spark, and then describe several important Spark-specific optimizations that we incorporated when implementing TIDE.

A. Spark Overview

Spark is a general-purpose distributed processing framework based on a functional programming paradigm. Spark provides a distributed memory abstraction called a Resilient Distributed Dataset (RDD) to support fault-tolerant computation across a cluster of machines. RDDs can either reside in the aggregate main memory of the cluster, or in efficiently serialized disk blocks. An RDD is immutable and cannot be modified, but a new RDD can be constructed by transforming an existing RDD. Spark utilizes both lineage tracking and checkpointing for fault tolerance.

B. Implementation and Optimization

Figure 4 demonstrates the end-to-end data pipeline of the TIDE system as implemented on Spark. First, TIDE leverages Spark Streaming to ingest batches of arriving edges, thereby supporting input sources such as HDFS, Kafka, Flume, and so on. The new edges are fed into the incremental updating component that maintains the sample graphs, the result of which is a compact in-memory RDD representing the aggregate graph of N samples. TIDE then extracts s samples (where s is the size of a bulk set) and transforms them into a partial aggregate graph in the Spark GraphX distributed graph RDD representation. The iterative bulk graph analysis algorithm is then executed on this representation of the partial aggregate graph. TIDE repeats the above process for the successive bulk sets of s sample graphs until all of the N sample graphs are analyzed. The result of each bulk graph analysis is an RDD that can be stored on HDFS or fed into various reporting tools.

Inside the incremental sample updating component, each batch of new edges is stored in an RDD b_t, where t is the time stamp, and the current aggregate graph is stored in an RDD g_t. Initially, the aggregate graph is just the first batch of edges, i.e. $g_0 = b_0$. At $t = 1$, a new RDD g'_0 is created from g_0 by applying a set of transformations that implement the one-step edge decay process (i.e., the Bernoulli subsampling step). Next, g'_0 is unioned with b_1 to produce the updated RDD g_1. This process continues as time advances. For eager incremental updating, the decay transformations include a map operation to update the bit array of each edge and a filter operation to discard edges that have become nonexistent in all sample graphs. For lazy incremental updating, the decay transformation comprises only a filter operation to check whether an edge has become nonexistent.

The bulk graph analysis component of TIDE is built on top of GraphX [7] which is an implementation of the Pregel and GraphLab processing frameworks on top of Spark. In particular, we implement a bulk execution wrapper for GraphX that performs the vertex-centric computation on the partial aggregate graph as in Algorithm 1.

Finally, we use the lineage and checkpointing mechanisms in Spark to support fault tolerance in TIDE. In what follows, we highlight some implementation optimizations that are specific to Spark.

1) In-Place Update: Because of their efficiency, the TIDE implementation uses memory-resident RDDs extensively. Memory management is a challenge, however. Because RDDs are immutable, TIDE must continuously create new RDDs as new edges arrive; indiscriminate creation of a large number of objects can quickly saturate memory. This is especially problematic for eager incremental updating, because of its higher memory requirement for storing the aggregate graph. Therefore, TIDE avoids creating new objects by applying in-place updates whenever possible. That is, new RDDs are still created, but they refer to existing objects in old RDDs. To keep the lineage of RDDs intact, TIDE must also notify Spark that the old in-memory RDDs have been changed. Thus, if an old RDD needs to be reprocessed (e.g., in case of a failure), it must first be regenerated from the latest checkpoints rather than being read directly from memory. In Section VII-A, we explore the effectiveness of in-place update for eager and lazy incremental updating methods.

2) Location-Aware Balancing Coalesce: In Spark, an RDD is divided into a set of partitions that are distributed among the workers in a cluster; each worker can have multiple partitions. Spark tracks the lineage of each partition, i.e., its parent partitions and the operations required to obtain the partition from its parents. For map and filter operations, the resulting RDD has exactly the same number of partitions as the parent RDD, even though some of the partitions can become empty after the filter operation. The union of two RDDs having k_1 and k_2 partitions is an RDD having $k_1 + k_2$ partitions; the partitions are simply unioned together.

In the incremental updating process, we need to repeatedly thin the aggregate graph through filter operations (and also map operations, in the case of eager updating) and union it with arriving edges. This procedure creates a potential problem. If the RDD for each batch contains k partitions, then, after ingesting n batches, the aggregate graph RDD would comprise nk partitions with highly skewed sizes. Indeed, a partition with older edges is likely to be quite small, or even empty. Since the partition serves as the basic scheduling unit in Spark, the presence of many small and empty partitions incurs a lot of unnecessary scheduling overhead.

Spark provides a *coalesce* operation to reduce the number of partitions in an RDD. If *shuffling*-based coalesce is used, then data in the RDD are reshuffled to generate fewer balanced partitions; otherwise, local partitions are simply merged together. Neither approach is directly applicable to our setting. Because coalesce needs to be applied frequently, shuffling is too expensive. On the other hand, arbitrary merging of local partitions yields highly imbalanced partition sizes. To avoid shuffling data while generating balanced partitions, we extend Spark with a *location-aware balancing coalesce* operation. This new coalesce operation combines local partitions (and thus avoids shuffling), but carefully chooses the candidate partitions based on their sizes by applying the Longest Processing Time (LPT) heuristic [25].

3) Distributed Monte Carlo Simulation: The eager incremental updating approach requires independent Bernoulli trials on each edge in each sample graph. To ensure that there is no correlation between the pseudorandom numbers generated for different Spark workers, we use the technique discussed in [26] for generating multiple streams of uniform numbers that are provably disjoint. In addition, we track the starting seed for each Spark partition, so that an updating operation on a given partition always produces exactly the same result if executed again (e.g., during failure recovery).

VII. Experimental Evaluation

In this section, we first describe some experiments designed to test the performance of our techniques for maintaining a set of sample graphs. We then evaluate the quality and performance of the PED approach when the sample graphs are used for influence analysis and community analysis.

Cluster Setup. All experiments were conducted on a cluster of 17 IBM System x iDataPlex dx340 servers. Each has two quad-core Intel Xeon E5540 2.8GHz processors and 32GB RAM; servers are interconnected using a 1Gbit

Ethernet. Each server runs Ubuntu Linux and Java 1.6. One server is dedicated to run the Spark coordinator and each of the remaining 16 servers is configured to run a Spark worker. We set SPARK_WORKER_CORES=8, SPARK_WORKER_MEMORY=28G, and default values for the other Spark parameters, based on standard practice.

Dataset. We used a real Twitter dataset for our experiments. It was obtained via the GNIP service and comprises 10% of the tweets generated between Sep 9, 2011 and Feb 29, 2012. We extracted the mention interactions out of this Twitter dataset and formed dynamic graphs. On average, 13.9 million new interactions were added per day. We experimented on daily batches, 2-day batches and 3-day batches in our empirical studies. The reason for such a coarse-grained discretization is to ensure that the data is of a large enough scale to test the system, since our dataset is only a small sample of the Twitter stream. In real settings, interactions are generated much more frequently, and thus a fine-grained discretization such as hourly batches would be adopted. In our experiments, the largest running aggregate graph contains around 65 million vertices and 1 billion edges.

Parameters. There are three important parameters that need to be specified in TIDE: the decay rate p, the total number of sample graphs N to incrementally maintain, and the number of sample graphs s in each bulk set for graph execution.

The decay rate p is completely application specific, and controls the proportion of historical interactions that an application considers in the analysis. For example, by setting $p = 0.8$, around 0.1% of the interactions from 30 periods ago are included in the current analysis. As another example, suppose that we want to ensure that, with probability $q = 0.01$, an influencer who had $n = 1000$ interactions $k = 60$ periods ago is still represented in the current network, where "represented" means having at least one adjacent edge remaining. Then we would set $p = [1 - (1 - q)^{1/n}]^{1/k} \approx 0.825$.

The number N of sample graphs controls the precision of the results. A variety of statistical methodologies are available for determining a good value of N. A comprehensive discussion of this topic is beyond the scope of the paper, so we content ourselves with a few examples. In the simplest setting, the goal of the analysis is to compute an expected value μ of an analytic graph function F with respect to the possible-graph distribution $\mathbb{P}_{f,t}$ defined in (1). That is, $\mu = \sum_{G' \in \mathcal{G}_t} F(G')\mathbb{P}_{f,t}(G')$ or, equivalently, $\mu = E[F(G')]$, where G' is a sample graph at time t. As an example, F might return the average influence score of the top 100 influencers. Given an initial value of N, we compute i.i.d. result samples $X_1, X_2, \ldots, X_N$, where $X_i = F(G_t^{f,i})$ and $G_t^{f,i}$ is the ith sample graph. Then $\hat{\mu}_N = N^{-1}\sum_{i=1}^{N} X_i$ is an unbiased and strongly consistent estimator of μ. Assuming that N is sufficiently large (say, $N \geq 20$), one can compute a standard $100(1-\delta)\%$ approximate confidence interval as $\hat{\mu} \pm z_\delta s_N/\sqrt{N}$, where z_δ is the $(1 - 0.5\delta)$-quantile of the standard normal distribution and s_N is the sample standard deviation of $X_1, X_2, \ldots, X_N$. If the confidence interval is too wide and the desired accuracy is $\pm 100\varepsilon\%$, then, going forward, N can be increased to $N^* = z_\delta^2 s_N^2/(\varepsilon\hat{\mu}_N)^2$ to try and achieve the desired accuracy. In general, we can monitor the confidence interval of the results as time progresses and increase N on the fly when the estimated accuracy falls below a threshold.[1] If F takes values in $\Re^d$ for some $d > 1$, then the above methodology can be applied, but using, e.g., an appropriate hyper-rectangular confidence region of specified maximum edge length on the d quantities of interest [27]. In more complex situations where, e.g., F returns a list of top-k influencers or an iceberg-query result of all persons with influence score above a threshold, simple averaging of the results from the different sample graphs may not suffice—see, e.g., [28]. The procedures for aggregating the results might then become complex, so that simple formulas for estimating error may not be available. In this case, bootstrapping techniques or other methods for assessing uncertainty may be needed; see [29] for a recent discussion.

As discussed in Section V-A, increasing the bulk set size s helps compression but also leads to higher memory requirements. Suppose that we have an upper bound on the expected amount of the memory occupied by a partial aggregate graph of s samples plus the memory consumed by the messages passed while processing this graph. A safe choice is to set s as large as possible such that the upper bound does not exceed the aggregate worker memory size. We have derived such a bound for our TIDE prototype, but omit the details because the bound is highly specific to our particular Spark implementation.

For the experiments in this section, we found that $p = 0.8$ is a reasonable decay rate for our example graph applications. Using the processes for choosing N and s as described above, we found that $N = 96$ provides accurate enough results for all experiments and $s = 16$ is a safe choice for the 3-day batch dataset and graph algorithms used in our experiments. However, in order to demonstrate the effect of the three parameters on the performance of TIDE, we also experiment with different settings of p, N and s.

In our experiments, we load the streaming input data as a sequence of HDFS files and produce an output sequence of HDFS files that represent the final analytic results at successive time points. We focus on evaluating the performance of the incremental updating and bulk graph analysis components of the TIDE system pipeline shown in Figure 4, because the time of the remaining operations (reading input, extracting partial aggregate graphs, and outputting analysis results) is negligible by comparison. Indeed, for the iterative graph algorithms we consistently observed that these remaining operations comprised less than 1.5% of the total execution time.

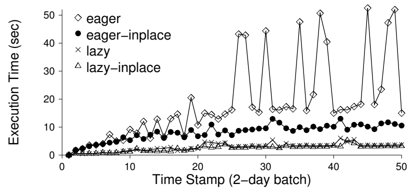


Fig. 5. Per-batch time for incremental updating

A. Incremental Updating Methods

In this section we empirically study the performance of incremental updating methods. The coalesce and checkpointing operations were carried out for every 10 batches. For the daily

[1]If N needs to be increased in TIDE, we have to compute the set of sample graphs from scratch. However, this happens infrequently.

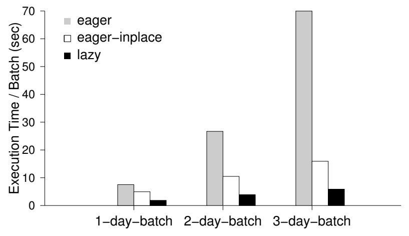


Fig. 6. Avg per-batch time for incremental updating (after 30th batch)

batch update, on average every checkpointing takes 47sec for eager updating but only 24sec for lazy updating, because the lazy updating method stores less data. We focus on per-batch comparisons between eager and lazy updating by excluding the times required for checkpointing and coalescing from the execution times reported below.

Comparison of Updating Methods. Figure 5 depicts the per-batch execution times for eager and lazy updating, both with and without in-place update, for the first 50 time stamps (batches), using the 2-day batch data as a representative. As shown, the in-place update has a huge effect on the eager updating method. This is because eager updating has a high memory requirement for storing the per-edge bit arrays. Without in-place updates, new bit arrays are continually created as decay transformations are applied. Indeed, a given edge can be re-created multiple times. When memory becomes saturated, garbage collection is invoked to reclaim obsolete objects, causing spikes on the curve for eager updating without in-place updates. In comparison, the lazy updating method benefits little from in-place update because it does not materialize bit arrays; we therefore omit the numbers for lazy updating with in-place updates in the remaining experiments. The running time for the naive sampling method is not reported in Figure 5 because it is extremely slow—it has to read and iterate over all the data at each batch arrival. For, e.g., the 50th batch, merely loading the data takes about 78 seconds, and extracting a single sample graph takes about 12 seconds, which means roughly 20 minutes are required for the naive method to obtain all of the sample graphs for this single batch.

It can be seen from Figure 5 that, for all four incremental updating methods, execution times initially grow as new edges are added, but gradually stabilize. This is because the aggregate graph size initially increases quickly, but the rate of increase tapers off by around the 30th batch, reflecting the probabilistic upper bounds discussed in Section IV-B.

Figure 6 displays, for various batch sizes, the average execution times per batch after the first 30 batches (i.e. after the execution times stabilize). In-place update shows increasing benefit—from 1.5x to 4.4x speedup—for eager updating as the batch size increases. Lazy updating exhibits a consistent speedup of approximately 2.7x over the in-place eager approach for the various batch sizes.

We also study the effects on system performance of the decay factor p and the number of samples N. Because of limited space, we only show results for the lazy updating method on 2-day batches, but experiments on other batch sizes exhibit the same trends. Table I displays the average execution time per batch under several different parameter settings. The running times increase in accordance with the

TABLE I. PER-BATCH TIME FOR LAZY UPDATING (AFTER 30TH BATCH)

parameter	avg time	# edges in 50^{th} batch
$p = 0.5, N = 96$	2.04 sec	201 million
$p = 0.8, N = 96$	3.87 sec	605 million
$p = 0.8, N = 192$	5.47 sec	683 million

TABLE II. COALESCE OPERATIONS FOR LAZY INCREMENTAL UPDATING

	shuffle-based	non-shuffle	location-aware
skewness	1.01	8.64	1.08
time (sec)	120.62	0.84	1.84

number of edges in the aggregate graph, but the increase is not necessarily proportional to the number of edges. This is because the incremental updating methods run very fast, so that Spark's job launching and task scheduling times become non-negligible.

Location-Aware Balancing Coalesce. We also study the impact of the location-aware balancing coalesce operation described in Section VI-B2 relative to the two existing shuffle-based and non-shuffle coalesce operations in Spark. We define the *skewness* of partitions as the ratio of the maximum partition size divided by the minimum partition size. The skewness is 1 for balanced partitions.

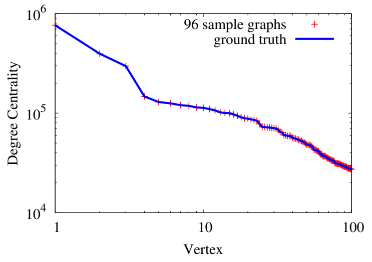
Table II compares the three coalesce operations when performed at the 40th time stamp of the lazy updating method using the 2-day batch dataset. Shuffle-based coalesce generates balanced partitions, but requires orders of magnitude more running time than the other methods. Non-shuffle coalesce is fast, but produces unacceptably skewed partition sizes. Our location-aware balancing coalesce produces good balanced partitions reasonably quickly. For eager updating, the skewnesses of the three coalesce operations are similar to those for lazy updating. The execution times for non-shuffle and location-aware balancing shuffle stay the same, since these two algorithms merely combine local partitions without touching the data underneath. However, the shuffle-based coalesce takes more time (350 sec) for eager updating.

B. Dynamic Graph Analysis

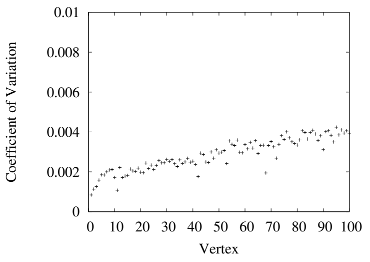
We choose three representative graph algorithms to demonstrate how our PED model can be used in two example graph applications. We then discuss the performance impact of the bulk graph execution technique. All experiments in this section were conducted on the 3-day batch datasets. To avoid dealing with the initial transient phase where graph size increases dramatically, we report qualitative results for the 40th batch and performance results from the 40th batch onward. The aggregate graph contains around 65 million vertices and 1 billion edges from the 40th batch onward.

1) Influence Analysis: Influence analysis is one of the most important types of analysis for social graphs. Centrality measures of vertices are widely used in practice for this application [30]. We chose the following two representative centrality measures:

Degree centrality. Degree centrality is the simplest way to measure the relative importance of a vertex in a graph. The degree-centrality score of a vertex v is defined as the number

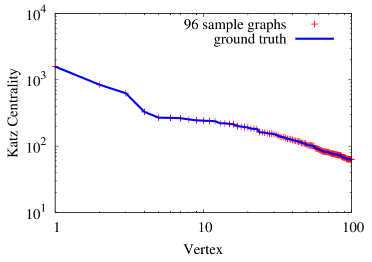


(a) Average vs ground truth

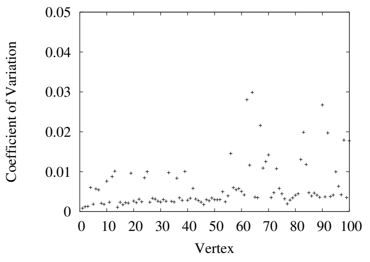


(b) Coefficient of variation

Fig. 7. Quality of results for degree centrality



(a) Average vs ground truth



(b) Coefficient of variation

Fig. 8. Quality of results for Katz centrality

of edges incident to v: $C_{\text{deg}}(v) = \sum_u |E_{(u,v)}|$, where $E_{(u,v)}$ is the set of edges from u to v.

Katz centrality. Katz centrality is a more complex measure of the importance of a vertex. The Katz centrality of a vertex v measures the number of paths that end at v, penalized by the path length: $C_{\text{Katz}}(v) = \sum_u \sum_{x \in Path_{(u,v)}} \alpha^{l(x)}$, where $Path_{(u,v)}$ is the set of paths connecting u to v, $l(x)$ is the length of path x, and α is an attenuation factor. In our experiments, we set $\alpha = 0.002$.

We consider analytic functions F that return the centrality score for each vertex in a graph, and our goal is to estimate μ, the expected value of F with respect to the possible-graph distribution $\mathbb{P}_{f,t}$. As discussed previously, we estimate μ unbiasedly by $\hat{\mu}_N = N^{-1} \sum_{i=1}^{N} F(G_t^{f,i})$; i.e., for each vertex, we compute the average centrality score over the N sample graphs at time t.

To evaluate the quality of the results, we compare the estimated influence scores in $\hat{\mu}_N$ to the ground-truth influence scores in μ. The ground-truth vector μ can be computed exactly by incorporating the decay probabilities in the centrality calculation. For degree centrality, the expected number of incident edges is computed as $C_{\text{deg}}(v) = \sum_u \sum_{e \in E_{(u,v)}} P^f(e)$, where $P^f(e) = f\big(t - t(e)\big)$ is the decay probability of e. For Katz centrality, the expected number of penalized paths connected to a vertex is computed as $C_{\text{Katz}}(v) = \sum_u \sum_{x \in Path_{(u,v)}} \alpha^{l(x)} P(x)$. Here $P(x)$ is the probability of a path x, which can be calculated as the product of the probabilities of the edges in x. Note that computing μ for either algorithm is prohibitively expensive for real-world applications, because it requires computation over all edges from the past. The size of the dynamic graph quickly grows beyond the capacity of any graph processing system.

Figures 7(a) and 8(a) compare, for each of the top 100 vertices (the 100 vertices with the highest true expected centrality scores), the average (over the 96 sample graphs) degree-centrality and Katz-centrality scores to the ground truth expected scores. As can be seen, the differences are almost indistinguishable.

Figures 7(b) and 8(b) display the coefficient of variation, over the 96 sample graphs, of centrality scores for the top 100 vertices. For degree centrality, all vertices have variations less than 0.5%, and for Katz centrality, although the variations are slightly higher, but they are all less than 3%. Clearly, the multiple samples yield good estimates of expected degree and Katz centralities. All of the above results show that choosing $N = 96$ achieves sufficient accuracy for both algorithms in our setting.

PED vs Snapshot and Sliding-Window Models. To demonstrate the potential practical benefits of the PED approach, we empirically compare the set of influencers (as measured by degree centrality) found from our real Twitter dataset when using a PED, snapshot, and sliding-window model (with a window size of three days). Among the top 100 influencers found by the PED model, about 24% of them were, like Alice in Example 1 of Section I, temporarily dormant influencers missed by the sliding-window model and 25% were, like Bob, rising star influencers missed by the snapshot model. Similarly, of the top 1000 influencers found by PED, 17% were like Alice and 26% were like Bob. In summary, a significant portion of potentially important influencers would be totally missed using either the snapshot model or the sliding-window model instead of PED.

2) Connectivity and Community Analysis: Connectivity and community analysis explores the community structure in social graphs. Existing studies [31] have shown that a social network usually contains a *giant* connected component that consists a constant fraction of the entire graph's vertices. In this example application, we study the characteristics of this giant component under the PED model.

We ran the connected-component algorithm on each sample graph to identify the giant component. The average size of the 96 giant components is 32.3 million vertices with a small standard deviation of only 2207. We observed that about 19 million vertices belong to the giant components of all sample graphs and form the high-probability "backbone" of the giant component. On the other hand, there are about 11.1 million vertices that appear in less than 10% of the sample graphs. Such vertices are connected to the network via edges that are infrequent and/or old. Our PED model can help us understand these two different types of vertices.

3) Performance of Bulk Graph Execution: In this subsection, we evaluate the performance of the bulk graph execution technique when analyzing degree centrality, Katz centrality, and connected-component structure. The three analysis algorithms span a range of graph analysis complexities. Determination of degree centrality does not require any iterative computation. The computation of Katz centrality is iterative, similar to that of PageRank. Moreover, it can use the incremental graph analysis scheme discussed in Section V-B to incrementally update the centrality scores from time t to time $t+1$. Connected-component computation is iterative but cannot leverage the incremental graph analysis scheme. This is because the label-propagation-based algorithm [32] cannot correctly handle incremental deletions of edges, so that recomputation from scratch is necessary at each time point.

In this experiment, we measure average bulk graph processing results from the 40th update onward, i.e., after stabilization. We use LZF compression for shuffling in Spark. Empirically, we observed that the use of LZF reduced run times by up to 46% for the bulk graph execution model and up to 12% for the naive execution model (processing one sample graph at a time).

Figure 9 compares the bulk graph execution model to the naive approach for the above three algorithms and for various values of the bulk-set size s. At any time point during the iterative Katz-centrality or connected-component algorithms, convergence occurs at roughly the same speed for all N sample graphs, due to their similar topologies and computation states. When processing the 40th batch, for example, the connected-component algorithm converges in 13 to 15 iterations for most sample graphs. Because the Katz-centrality computation takes roughly the same amount of time for each iteration, we report the per-iteration execution time, whereas we report the total execution time for the other two algorithms.

The bulk graph execution model essentially degenerates to the naive approach when $s = 1$, but the execution times are all slower than that of the naive approach, due to the overhead of the bulk-execution wrapper. As s increases, this overhead is quickly amortized and the per-sample-graph performance gradually surpasses that of the naive approach. However, at some point, the advantage starts to decrease due to the higher memory burden of storing a larger partial aggregate graph. Figure 9(b) also shows the running time for Katz centrality for different decay factors p. The running time under $p = 0.5$ is significantly less than $p = 0.8$ because the aggregation graph contains only about one third of the edges. Still, bulk execution significantly reduces the average running time per sample graph in both cases.

Bulk graph execution benefits the simple degree centrality algorithm much more than the two iterative algorithms, because a non-trivial overhead must be paid per iteration in GraphX. Consider, for example, the execution times for the first five iterations of the connected-component algorithm; see Figure 9(c). It is known [32] that most of the computation in this algorithm occurs in the first few iterations (five iterations in our case). Even though there is very little to do in remaining iterations, we still must pay the per-iteration overhead in GraphX. As can be seen, the time for each remaining iteration is more or less the same for different bulk-set sizes.

As discussed before, the iterative Katz-centrality algorithm is able to leverage the incremental graph analysis scheme by using the end states at a time point as the starting states for the next time point. Empirically, we observed substantial performance improvements when using this incremental scheme. As an example, for a randomly chosen sample graph at the 40th batch, the Katz-centrality algorithm requires 28 iterations to converge if computing from scratch. In contrast, by reusing the result of the 39th batch, only four iterations are needed.

VIII. Related Work

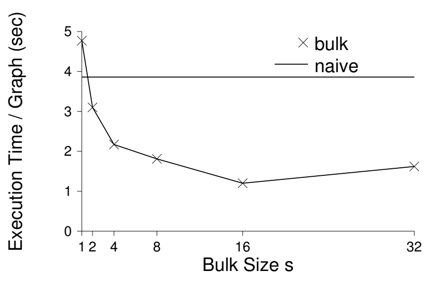
In recent years, a number of distributed graph processing systems [17], [33], [34], [19], [20], [35], [32] have been proposed for static graphs. For distributed processing of dynamic graphs, existing systems include Kineograph [1], as well as environments designed for incremental iterative data flows, such as Naiad [2] and the system described in [24]. All three of these systems, however, are based on the snapshot model. Several recent works have investigated, from a graph-database perspective, the problems of storing and retrieving large-scale evolving graphs [3], [36], [37], but they do not consider complex graph analytics such as influence-analysis and community-detection algorithms.

In [38], a modified definition of Katz centrality is proposed to capture both time-dependency and recency of random walks in a dynamic graph. In comparison, TIDE is not designed for a specific graph algorithm, but rather is a general platform that allows direct application of a wide range of static graph algorithms to dynamic graphs.

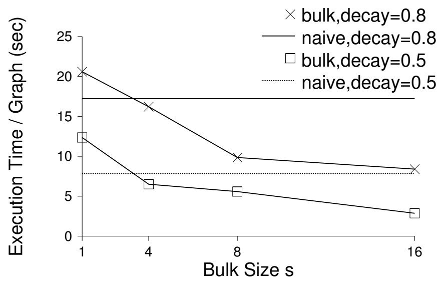
The general idea of temporally biased sampling in (non-graph) data streams was introduced in [4] to reduce staleness in the sample in order to obtain analytic results more relevant to the present. Data-decay methods were studied for data streams [39], but the focus was on relatively simple aggregation queries. The use of probabilistic edge decay for analysis of dynamic graphs has not been formally studied before.

IX. Conclusion

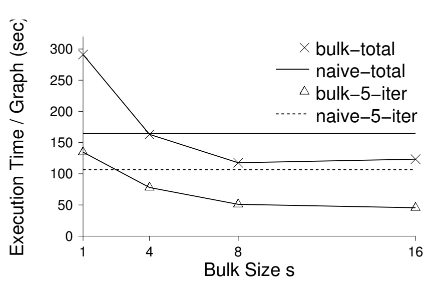
We have described TIDE, a distributed system for analyzing dynamic graphs. TIDE employs a model based on probabilistic edge decay to implement a temporally biased sampling scheme that allows controlled trade-offs between emphasizing recent interactions and providing continuity with respect to past interactions; the PED model generalizes existing snapshot and sliding-window models. To facilitate maintenance of a set of sample graphs, we have provided both provably compact "aggregate" representations and efficient incremental updating methods. We have also introduced a bulk execution model to simultaneously process these graphs using the same programing APIs as are found in existing static graph processing systems. Through experiments on a Twitter dataset, we have demonstrated the effectiveness and efficiency



(a) Degree centrality (total)



(b) Katz centrality (per iteration)



(c) Connected component (total)

Fig. 9. Bulk graph execution vs the naive approach

of our proposed methods for tasks such as identifying key influencers and exploring community structure. Future work includes investigating decay functions beyond the exponential function and leveraging results from the probabilistic database community to obtain a more comprehensive set of analysis techniques for sample graphs.

References

[1] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen, "Kineograph: taking the pulse of a fast-changing and connected world," in *EuroSys*, 2012.

[2] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard, "Differential dataflow," in *CIDR*, 2013.

[3] J. Mondal and A. Deshpande, "Managing large dynamic graphs efficiently," in *SIGMOD*, 2012.

[4] C. C. Aggarwal, "On biased reservoir sampling in the presence of stream evolution," in *VLDB*, 2006.

[5] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, 2012.

[6] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *SOSP*, 2013.

[7] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *OSDI*, 2014.

[8] D. M. Romero, B. Meeder, and J. Kleinberg, "Differences in the mechanics of information diffusion across topics: Idioms, political hashtags, and complex contagion on twitter," in *WWW*, 2011.

[9] M. Roth, A. Ben-David, D. Deutscher, G. Flysher, I. Horn, A. Leichtberg, N. Leiser, Y. Matias, and R. Merom, "Suggesting friends using the implicit social graph," in *KDD*, 2010.

[10] P. S. Yu, X. Li, and B. Liu, "On the temporal dimension of search," in *WWW Alt*, 2004.

[11] L. Zheng, C. Shen, L. Tang, T. Li, S. Luis, and S.-C. Chen, "Applying data mining techniques to address disaster information management challenges on mobile devices," in *KDD*, 2011.

[12] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang, "Timestream: reliable stream computation in the cloud," in *EuroSys*, 2013.

[13] A. Volkova, "A refinement of the central limit theorem for sums of independent random indicators," *Theory Probab. Appl.*, vol. 40, no. 4, pp. 791–794, 1996.

[14] "Murmurhash," sites.google.com/site/murmurhash.

[15] P. L'Ecuyer and R. Simard, "Testu01: A c library for empirical testing of random number generators," *ACM Trans. Math. Softw.*, vol. 33, no. 4, 2007.

[16] A. M. Law, *Simulation Modeling and Analysis*, 5th ed. McGraw-Hill, 2014.

[17] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *SIGMOD*, 2010.

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

[19] B. Shao, H. Wang, and Y. Li, "Trinity: A Distributed Graph Engine on a Memory Cloud," in *SIGMOD*, 2013.

[20] G. Wang, W. Xie, A. Demers, and J. Gehrke, "Asynchronous large-scale graph processing made easy," in *CIDR*, 2013.

[21] L. Katz, "A new status index derived from sociometric analysis," *Psychometrika*, vol. 18, no. 1, pp. 39–43, 1953.

[22] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *WWW*, 1998.

[23] D. Eppstein, Z. Galil, and G. F. Italiano, *Dynamic Graph Algorithms*. CRC Press, 1999.

[24] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl, "Spinning fast iterative data flows," *PVLDB*, vol. 5, no. 11, 2012.

[25] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969.

[26] H. Haramoto, M. Matsumoto, T. Nishimura, F. Panneton, and P. L'Ecuyer, "Efficient Jump Ahead for 2-Linear Random Number Generators," *INFORMS Journal on Computing*, vol. 20(3), pp. 385–390, 2008.

[27] R. G. Miller, *Simultaneous Statistical Inference*, 2nd ed. Springer, 1981.

[28] I. F. Ilyas and M. A. Soliman, *Probabilistic Ranking Techniques in Relational Databases*, ser. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[29] S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, M. I. Jordan, S. Madden, B. Mozafari, and I. Stoica, "Knowing when you're wrong: building fast and reliable approximate query processing systems," in *SIGMOD*, 2014, pp. 481–492.

[30] S. P. Borgatti and M. G. Everett, "A graph-theoretic perspective on centrality," *Social Networks*, vol. 28, no. 4, pp. 466–484, 2006.

[31] P. S. Yu, J. Han, and C. Faloutsos, Eds., *Link Mining: Models, Algorithms, and Applications*. Springer, 2010.

[32] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From "think like a vertex" to "think like a graph"," *PVLDB*, vol. 7, no. 3, 2013.

[33] "Apache Giraph," giraph.apache.org.

[34] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *OSDI*, 2012.

[35] W. Xie, G. Wang, D. Bindel, A. Demers, and J. Gehrke, "Fast iterative graph computation with block updates," *PVLDB*, vol. 6, no. 14, 2013.

[36] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng, "On querying historical evolving graph sequences," *PVLDB*, vol. 4, no. 11, 2011.

[37] U. Khurana and A. Deshpande, "Efficient snapshot retrieval over historical graph data," in *ICDE*, 2013.

[38] P. Grindrod and D. J. Higham, "A matrix iteration for dynamic network summaries," *SIAM Rev.*, vol. 55, no. 1, pp. 118–128, 2013.

[39] E. Cohen and M. J. Strauss, "Maintaining time-decaying stream aggregates," *J. Algorithms*, vol. 59, no. 1, pp. 19–36, 2006.