Scaling Out Link Prediction with SNAPLE: 1 Billion Edges and Beyond

Anne-Marie Kermarrec
INRIA Rennes, France
anne-marie.kermarrec@inria.fr
Francois Taiani
U. of Rennes 1 - IRISA - ESIR
Rennes, France
ftaiani@irisa.fr
Juan M. Tirado
University of Cambridge
Cambridge UK
jmt78@cl.cam.ac.uk

ABSTRACT
A growing number of organizations are seeking to analyze extra large graphs in a timely and resource-efficient manner. With some graphs containing well over a billion elements, these organizations are turning to distributed graph-computing platforms that can scale out easily in existing data-centers and clouds. Unfortunately such platforms usually impose programming models that can be ill suited to typical graph computations, fundamentally undermining their potential benefits.

In this paper, we consider how the emblematic problem of link-prediction can be implemented efficiently in gather-apply-scatter (GAS) platforms, a popular distributed graph-computation model. Our proposal, called SNAPLE, exploits a novel highly-localized vertex scoring technique, and minimizes the cost of data flow while maintaining prediction quality. When used within GraphLab, SNAPLE can scale to very large graphs that a standard implementation of link prediction on GraphLab cannot handle. More precisely, we show that SNAPLE can process a graph containing 1.4 billions edges on a 256 cores cluster in less than three minutes, with no penalty in the quality of predictions. This result corresponds to an over-linear speedup of 30 against a 20-core standalone machine running a non-distributed state-of-the-art solution.

Keywords
link prediction, graph processing, parallelism, distribution, scalability

1. INTRODUCTION

Graph computing is today emerging as a critical service for many large-scale on-line applications. Companies such as Twitter, Facebook, and LinkedIn are capturing, storing, and analyzing increasingly large amounts of connected data

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 2015 ACM 978-1-4503-3618-5 ...$15.00.

stored as graphs. As the size of these graphs increases, these companies are moving away from standalone one-machine deployments [12] and are instead looking for distributed solutions [27, 25] that can harvest the resources of multiple machines to process these graphs in parallel. Distribution unfortunately comes with an extra complexity, which can considerably hamper a solution’s scalability if not properly managed.

In this paper, we focus on the particular problem of link-prediction [22] in large graphs, an emblematic graph analysis task that appears in numerous applications (content recommendations, advertising, social mining, forensics). Implementing link prediction on a distributed graph engine raises two critical challenges: (1) traditional link prediction approaches are ill-fitted to the programming models of graph processing engines; (2) because of this bad fit communication costs can be difficult to keep under control, reducing the benefits of distribution.

More precisely, the link prediction problem considers a graph \( G \) in which some edges are missing, and tries to predict these missing edges. To be able to scale, most practical solutions search for missing edges in the vicinity of individual vertices, using bounded graph traversal techniques such as bounded random walks, or \( d \)-hops neighborhood traversal.

Unfortunately, these graph traversal techniques requires large amounts of information to be propagated between vertices, and do not lend themselves to the highly localized models offered by distributed graph engines, such as the Bulk Synchronous Processing (BSP) model of Pregel [27], or the Gather Apply Scatter (GAS) model of GraphLab [25]. In both cases, a naive application of traversal techniques requires vertex information to be replicated to maintain locality, and can lead to high communication and memory costs. This is particularly true of graphs in which the likelihood of two vertices being neighbors can be computed easily, but which require large amounts of information to be shared between vertices, such as in social graphs with large user profiles.

In this paper, we propose a highly scalable approach to link-prediction that can be implemented efficiently within 1 distributed graph engines for short
the Gather Apply Scatter (GAS) model. The resulting system, called SNAPLE, relies on a scoring framework of potential edges that eschews large data flows along graph edges. Instead, SNAPLE combines and aggregates similarity scores along the paths of the original graph, and thus avoids explicit and costly graph traversal operations. We demonstrate the benefits of SNAPLE with a prototype based on Graphlab [25], which we evaluate using real datasets deployed on top of a testbed with 32 nodes and 256 cores. Our experiments show that SNAPLE’s performance goes well beyond that of a standard GAS implementation, and is able to process a graph containing 1.4 billions edges in 2min57s on our testbed, when a naive GraphLab version fails due to resource exhaustion. We obtain these results with no penalty in prediction quality compared to a traditional approach. SNAPLE further demonstrates linear or over-linear speedups in computation time against a single machine deployment running a state-of-the-art nondistributed solution while improving predictions.

In the following we first describe link-prediction in more detail, and introduce the Gather Scatter Apply (GAS) programming model (Section 2). In Section 3, we present the principles of SNAPLE, our link prediction framework. We then detail how SNAPLE can be implemented efficiently on a GAS engine in Section 4. Section 5 presents an exhaustive evaluation of our approach. Finally, Section 6 discusses related work, and Section 7 concludes.

2. BACKGROUND AND PROBLEM

2.1 Link-prediction.

Link prediction [22] seeks to predict missing edges from a graph $G$. Edges might be missing because the graph is evolving over time (users create new social links), or because $G$ only captures a part of a wider ground truth. Predicted edges can be used to recommend new users (social graphs), new items (bipartite graph), or uncover missing information (social mining).

Formally, link prediction considers two graphs\(^2\) $G = (V, E)$ and $G' = (V, E')$ so that $G$ contains less information than $G'$ in the form of missing edges: $E \subseteq E'$. For instance $G$ and $G'$ might represent the same social graph captured at different points in time, or $G$ might represent an incomplete snapshot of a larger graph represented by $G'$ (an interaction network, a set of related topics, etc.). The goal of link prediction is then to predict which are the edges of $G'$ that $G$ lacks, i.e. to determine $E' \setminus E$.

Link prediction strategies fall into unsupervised and supervised approaches. The typical approach for unsupervised link prediction is sketched in Algorithm 1. The algorithm executes a parallelizable loop that iterates through all vertices of $G$ (lines 1-3). Each iteration hinges on the scoring function $\text{score}(u, z)$ (line 2) which reflects how likely the edge $(u, z)$ is to appear in $G'$. In this basic version, the algorithm scores all vertices $z$ that are not already in $u$’s neighborhood (noted $\Gamma_G(u) = \{v \in V | (u, v) \in E\}$), and returns the $k$ vertices with the highest scores as predicted neighbors for $u$ (operator $\text{argtop}^k$).

The function $\text{score}(u, z)$ may only use topological proper-

\(^2\)For brevity’s sake, we consider directed graphs in our explanations, but the same principles carry over to undirected graphs.

Algorithm 1 Unsupervised top $k$ link-prediction

Require: $k$, score
1: for $u \in V$ do
2: $\text{prediction}_u \leftarrow \text{argtop}^k(\text{score}(u, z))$ zE\V \leftarrow \text{Gamma}^k(\text{score}(u, z))$
3: end for

ties, such as the connectivity-based metrics proposed in the seminal work of Liben-Nowell and Kleinberg [22] (e.g. the number of common neighbors between $u$ and $z$, $|\Gamma_G(u) \cap \Gamma_G(z)|$). This score may also exploit vertex content, i.e. the additional application-dependent knowledge attached to vertices [7, 16, 31, 32, 39], such as user profiles, tags, or documents. In many domains, pure topological metrics tend to be the main drivers of link generation, and are therefore almost always present in the prediction process. Scoring functions using only topological metrics are also more generic as they do not rely on information that is external to the graph (tags, content, user profiles, etc.) and might not be available in all data sets.

Supervised approaches build upon unsupervised strategies and leverage machine-learning algorithms to produce optimized scoring functions [23, 38]. Supervised approaches tend to perform better, but at the cost of an important learning effort, as they must often scan the whole graph to build an accurate classification model. In this paper, we therefore focus on unsupervised approaches, but the key ideas we present can be extended to supervised schemes.

2.2 Scaling link prediction

While research on link prediction originally sought to maximize the quality of predicted edges, with little consideration for computation costs, its practical relevance for social networks and recommendation services has put it at the forefront of current system research. One critical challenge faced by current implementations is the fast growing size of the graphs they must process [12, 21].

A first strategy to scale Algorithm 1 is to improve the performance of individual iterations. A frequent optimization limits the search for missing edges to the vicinity of individual vertices. If one notes $\Gamma_G^K(u)$ the $K$-hop neighborhood of $u$ in $G$, defined recursively as

\[
\begin{align*}
\Gamma_G^1(u) &= \Gamma_G(u) \\
\Gamma_G^K(u) &= \Gamma_G^{K-1}(u) \cup \{z \in V | \exists v \in \Gamma_G^{K-1}(u) : (v, z) \in E\}
\end{align*}
\]

this optimization will only consider the vertices of $\Gamma_G^K(u) \setminus \Gamma_G(u)$ at line 2 as potential new neighbors in $G'$ (Equation 2 below), instead of the much larger set $V \setminus \Gamma_G(u)$. $K$ is generally small (2 or 3). We use $K = 2$ in this work.

\[
\text{prediction}_u \leftarrow \text{argtop}^k(\text{score}(u, z))
\]

This optimization works well because social graphs, and field graphs in general, tend to present high clustering coefficients. As a result, most of the edges to be predicted in $G'$ will connect vertices only separated by a few hops in $G$ [44]. Other optimizations on standalone machines leverage specialized data structures and memory layout to exploits data locality and minimize computing costs [20, 33].

A second strategy seeks to scale Algorithm 1 horizontally by deploying it onto a distributed infrastructure. That is the case of Twitter for instance, who recently moved their
Who-to-Follow service to a distributed solution, from an initial single machine deployment [12]. This transition can leverage a growing number of graph processing engines [5, 6, 12, 15, 17, 26, 27, 34], which aim to facilitate the realization of scalable graph processing tasks. These engines do so by implementing highly parallelizable programming models such as Bulk Synchronous Processing (BSP) [17, 27, 34], or Gather, Apply, Scatter (GAS) [25].

2.3 The GAS model

In this work, we focus particularly on the GAS model, which can be seen as a refinement of map-reduce and BSP for graphs. Its reference implementation, GraphLab [11, 25], is particularly scalable, and was found to perform best in a recent comparison of modern graph engines across a number of typical graph computing tasks [13].

More precisely, a GAS program assumes every vertex \( u \in V \) and every edge \((u, v) \in E\) of a graph \( G = (V, E)\) is associated with some mutable data, noted \( D_u \) and \( D_{(u,v)} \). A GAS program consists of a sequence of GAS super-steps (or steps). Each step comprises three conceptual phases that execute in parallel at each vertex and manipulate these data. (Our notation follows closely that of [11].)

1. The gather phase is similar to a map-and-reduce step. This phase collects the data associated with a vertex \( u \)'s neighbors \( \Gamma_C(u) \) and with \( u \)'s out-going\(^3\) edges \((u) \times \Gamma_C(u)\). It maps this data through a user-provided gather() function; and aggregates the result using a general sum() function defined by the user.

\[
\Sigma \leftarrow \text{sum} \{ \text{gather}(D_u, D_{(u,v)}, D_v) \} \quad (3)
\]

2. In the apply phase, the result of the gather phase, \( \Sigma \), is used to update the current data of node \( u \), \( D_u \), using a user-defined apply function.

\[
D'_u \leftarrow \text{apply}(\Sigma, D_u) \quad (4)
\]

3. Finally the scatter phase uses \( \Sigma \) and the new value of \( D_u \) to propagate information within \( u \)'s neighborhood using a scatter function provided by the user.

\[
\forall v \in \Gamma_C^{-1}(u) : D'_{(v,u)} \leftarrow \text{scatter}(\Sigma, D'_u, D_{(v,u)}) \quad (5)
\]

2.4 Link prediction in the GAS model

The GAS model facilitates the scheduling and parallelization of vertex operations while increasing content locality. GAS engines, however, can require some substantial effort to adapt existing graph algorithms, for two reasons. First, graph traversals, a primitive strategy of many graph algorithms, are difficult to express in the GAS paradigm without adding substantial complexity and overhead. This is because the accesses and updates of a GAS step are limited to adjacent vertices and edges.

The second difficulty pertains to the limited access to topological information offered by the GAS paradigm. In the GAS model, vertices drive and organize the computation (principally in the gather and scatter phases), but are not expected to be an object of computation per se, in order to deal with densely connected vertices in power-law distributions. Vertices and neighborhoods are therefore not exposed directly as accessible data.

These two limitations are particularly problematic for the link prediction algorithm captured in equation (2). This algorithm traverses all vertices in the K-hop neighborhood \( \Gamma_K(u) \) of individual vertices, and requires access to topological information in its ‘score()’ function (see Figure 1). Choosing a naive approach to work around these limitations, by propagating state, and replicating data across vertices, can be extremely counter-productive, and tend to self-defeat the design rationale of the GAS paradigm.

Figure 1: Example of data propagation when computing score\((a, c)\), score\((a, d)\) and score\((a, e)\) with a naive GAS approach deployed on three computing nodes \((N_{0,1,2})\). We have to propagate \( \Gamma_C(c) \), \( \Gamma_C(d) \) and \( \Gamma_C(e) \) through \( b \) and \( f \) (dashed blue arrows). Then, this data has to be transferred to vertex \( a \), on a different machine \((N_1)\), and replicated onto the replicas of \( a \) (here on \( N_2 \)) for synchronization.

3. THE SNAPLE FRAMEWORK

To address the above challenges, we have developed a novel and lightweight framework that computes a similarity score \( \text{score}(u, z) \) between two vertices \( u \) and \( z \) without the costs associated with traditional similarity metrics. Our framework avoids a direct and costly computation of similarity by propagating intermediate results along the paths of the graph \( G \). This propagation involves two low-overhead steps: in a first path-combination step we combine raw similarity values along 2-hop paths in \( G \), resulting in a path-similarity for each 2-hop paths connecting \( u \) to \( z \). In a second path-aggregation step, we then aggregate these path-similarity values to compute score\((u, z)\). These two steps are configurable with user-provided functions for combination and aggregation, and provide in effect a configurable scoring framework tailored to the GAS paradigm.

3.1 Path-combination

Our approach starts from a “raw” similarity metric, that we use as a basic building block to construct our scoring framework. We consider topological metrics that can be computed from the neighborhoods of the two nodes \( u \) and \( z \) one wishes to compare:

\[
\text{sim}(u, z) = f(\Gamma(u), \Gamma(z)) \quad (6)
\]

where \( f \) is a similarity metric on sets, for instance Jaccard's coefficient [35]. This approach can be extended to content-based metrics [14] by including content data in \( f \).
In the following, we limit ourselves to the 2-hop neighborhood of \( u \) when searching for candidate nodes, i.e., we use \( K = 2 \) in (2), a typical value for this parameter. As explained earlier, a first challenge when directly using the similarity shown above in the GAS model to implement the score\((u, z)\) function of equation (2), is the inherent difficulty to access data attached to the nodes \( z \in \Gamma^2(u) \setminus \Gamma(u) \), which are not direct neighbors of \( u \). One naive solution consists in using an initial GAS step to propagate a node’s information to its neighbors, and make this data accessible to neighbors of neighbors in subsequent steps.

\[
D'_u, \text{neighborhood} \leftarrow \{ (v, D_v) \mid v \in \Gamma(u) \} \tag{7}
\]

Unfortunately, and as we will show in our evaluation, the redundant data transfer and additional storage this approach causes make it highly inefficient, yielding counterproductive results in particular on very large graphs.

In order to overcome this limitation, SNAPLE uses a path-combination step that returns a similarity value (termed path-similarity) for each 2-hop path \( u \to v \to z \) connecting a source vertex \( u \) with a candidate vertex \( z \):

\[
sim^*_p(u, z) = \sim(u, v) \odot \sim(v, z) \tag{8}
\]

where \( \odot \) is a binary operator that is monotonic-increasing on its two parameters, such as a linear combination, or generalized means. We call the operator \( \odot \) a combinator\(^4\).

Intuitively, equation (8) seeks to capture the homophily often observed in field graphs: if \( u \) is similar to \( v \) and \( v \) to \( z \), then \( u \) is likely to be similar to \( z \).

Table 1 lists examples of combiners that we consider in this work: a linear combination, the Euclidean distance, the geometric mean, a plain sum (a special case of linear combination), and a basic counter (a degenerated case where all similarity values are stuck to 1).

### 3.2 Path-aggregation

The path-combination step we have just described provides a similarity value \( \sim^*_p(u, z) \) for each 2-hop path \( u \to v \to z \) connecting \( u \) to \( z \). Multiple such paths are however likely to exist. For instance, in Figure 2, 2 paths can be reached from \( u \) over two 2-hop paths: \( u \to v \to z \) and \( u \to f \to z \), delivering two different similarity values, \( \sim^*_p(u, z) \) and \( \sim^*_p(u, z) \). SNAPLE aggregates these different values to obtain the final score of a node \( z \in \Gamma^2(u) \setminus \Gamma(u) \):

\[
\text{score}(u, z) = \bigoplus_{v \in \Gamma(u)^2 \setminus \Gamma^1(z)} \sim^*_p(u, z) \tag{9}
\]

where \( \Gamma^1(z) \) is the inverse neighborhood of \( z \), and \( \bigoplus \) is a multinary operator that can be decomposed in a generalized

\(^4\)We limit ourselves to 2-hop paths, but this approach can be extended to longer paths by recursively applying \( \odot \) to the raw similarities of individual edges (in functional terms, essentially executing a fold operation on the raw similarity values along the path connecting \( u \) to \( z \)).

---

**Figure 2:** Two paths connect \( u \) to \( z \). Both paths yield different similarity values, \( \sim^*_p(u, z) \) and \( \sim^*_p(f, z) \), which SNAPLE summarizes with an aggregator (\( \oplus \)).

---

**Table 1: Examples of combiners \( \otimes \)**

<table>
<thead>
<tr>
<th>name</th>
<th>( a \otimes b )</th>
<th>( \text{sim}^*_p(u, z) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear</td>
<td>( \alpha a + (1 - \alpha)b )</td>
<td>( \alpha \sim(u, v) + (1 - \alpha)\sim(v, z) )</td>
</tr>
<tr>
<td>eucl</td>
<td>( \sqrt{a^2 + b^2} )</td>
<td>( \sqrt{\sim(u, v)^2 + \sim(v, z)^2} )</td>
</tr>
<tr>
<td>geom</td>
<td>( a \times b )</td>
<td>( \sim(u, v) \times \sim(v, z) )</td>
</tr>
<tr>
<td>sum</td>
<td>( a + b )</td>
<td>( \sim(u, v) + \sim(v, z) )</td>
</tr>
<tr>
<td>count</td>
<td>( 1 )</td>
<td>( 1 )</td>
</tr>
</tbody>
</table>

**Table 2: Examples of aggregators \( \oplus \)**

<table>
<thead>
<tr>
<th>name</th>
<th>( a \oplus b )</th>
<th>( \oplus_{post}(x, n) )</th>
<th>( \oplus_{pre} \times X )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sum</td>
<td>( a + b )</td>
<td>( \frac{</td>
<td>x</td>
</tr>
<tr>
<td>Mean</td>
<td>( a + b )</td>
<td>( \frac{1}{n} \sum_{i \in X} x_i )</td>
<td>( \sum_{x \in X} )</td>
</tr>
<tr>
<td>Geom</td>
<td>( a \times b )</td>
<td>( a^{\frac{1}{n}} \sum_{i \in X} x_i )</td>
<td>( \sum_{x \in X} )</td>
</tr>
</tbody>
</table>

**Figure 3:** Illustrates this
impact when using the linear combinator ($\alpha = 0.5$) with the three aggregators of Table 2. In this example, $a$ is connected to $e$ and $f$ through two 2-hop paths, and to $g$ through three 2-hop paths. The highest score obtained with each aggregator is shown in bold in table.

The Mean and Geom aggregators (two bottom lines in the table), average out the number of paths connecting $a$ to each candidate vertices, and as a result, consider $f$ the vertex most similar to $a$. By contrast, the Sum aggregator takes into account the connectivity of candidate vertices in its final score, and rates $g$ over $f$ (first line). This means a vertex with lower path-similarities (such as $g$ here) can obtain a high final score if enough paths connect it to the source vertex ($a$). Finally, let us note how the Geom aggregator penalizes vertices such as $e$ which are connected through paths with very low path-similarity (here $a \rightarrow h \rightarrow e$).

4. IMPLEMENTING SNAPLE IN GAS

The approach we have just presented addresses the challenges inherent to the GAS model that we discussed in Section 2.4. Even using SNAPLE’s vertex score however, the candidate space $\Gamma^2(u) \setminus \Gamma(u)$ of equation (2) remains generally too large to be fully explored. Indeed, if we note $n$ the average out-degree of vertices in $G$, a blind application of (2) can require up to $O(n^2)$ scores to be computed per vertex on average, leading to $O(|V|n^2)$ scoring operations for the whole graph $G = (V, E)$.

This challenge is not inherent to the GAS model, but must be addressed in our implementation to obtain a tractable solution. We attack it using a two-pronged approach: first, we truncate neighborhoods that are larger with the highest similarity values at individual vertices. By contrast, the $aggregators (two bottom lines in (5)) scores to be computed per vertex on average, leading to $O(|V|n^2)$ scoring operations for the whole graph $G = (V, E)$.

Table 3: Examples of $score(u, z)$ combinations in SNAPLE’s scoring framework

<table>
<thead>
<tr>
<th>sim(u,v)</th>
<th>combinator (⊙)</th>
<th>aggregator (⊕)</th>
<th>score name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jaccard</td>
<td>linear</td>
<td>linearSum</td>
<td></td>
</tr>
<tr>
<td>Jaccard</td>
<td>eucl</td>
<td>euclSum</td>
<td></td>
</tr>
<tr>
<td>Jaccard</td>
<td>geom</td>
<td>Sum</td>
<td>PPR</td>
</tr>
<tr>
<td>1/</td>
<td>V</td>
<td></td>
<td>count</td>
</tr>
<tr>
<td>Jaccard</td>
<td>linear</td>
<td>linearMean</td>
<td></td>
</tr>
<tr>
<td>Jaccard</td>
<td>eucl</td>
<td>Mean</td>
<td>euclMean</td>
</tr>
<tr>
<td>Jaccard</td>
<td>geom</td>
<td>geomMean</td>
<td></td>
</tr>
<tr>
<td>Jaccard</td>
<td>linear</td>
<td>geomSum</td>
<td></td>
</tr>
<tr>
<td>Jaccard</td>
<td>eucl</td>
<td>Geom</td>
<td>euclGeom</td>
</tr>
<tr>
<td>Jaccard</td>
<td>geom</td>
<td>geomGeom</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3: Example of scores obtained using different aggregators and the linear combinator ($\alpha = 0.5$). Edge labels show the similarity between vertices.

Algorithm 2 SNAPLE’s link-prediction as a GAS program

Require: $k$, $sim$, $\oplus^pre$, $\oplus^{post}$, $klocal$, $thr^\Gamma$

1: gather$_1(D_u, D_{(u,v)}, D_v)$:
2: $γ \leftarrow \{v\}$
3: if rand() $> thr^\Gamma/|\Gamma(u)|$ then $γ \leftarrow \emptyset$
4: return $γ$
5: sum$_1(γ_1, γ_2)$: return $γ_1 \cup γ_2$ $\triangleright$ Computing $Σ_1$
6: apply$_1(D_u, Σ_1)$: $D_u.Γ \leftarrow Σ_1$

7: gather$_2(D_u, D_{(u,v)}, D_v)$:
8: return $\{(v, sim(u,v)) \equiv f(D_u, Γ, D_v, Γ)\}$
9: sum$_2(γ_1, γ_2)$: return $γ_1 \cup γ_2$ $\triangleright$ Computing $Σ_2$
10: apply$_2(D_u, Σ_2)$:
11: $D_u.\mathit{sims} \leftarrow \mathit{argtop}^{klocal}(s_v)$

$\triangleright$ Step 2: Estimate similarities

12: gather$_3(D_u, D_{(u,v)}, D_v)$:
13: if $v \not\in D_u.\mathit{sims}.\mathit{keys}$ then return $\emptyset$
14: $Γ_u \leftarrow D_u.Γ_u$ ; $Γ_v \leftarrow D_v.\mathit{sims}.\mathit{keys}$
15: return $\{(z, D_u.\mathit{sims}[v] \oplus D_v.\mathit{sims}[z], 1) \mid z \in Γ^\mathit{max}_{klocal}(u)\}$
16: sum$_3(γ_1, γ_2)$: return $\mathit{merge}(\oplus^pre, γ_1, γ_2)$ $\triangleright$ Comp. $Σ_3$
17: apply$_3(D_u, Σ_3)$:
18: $\mathit{score} \leftarrow \emptyset$
19: for $(z, s^\mathit{pre}, n_z) \in Σ_3$ do $\mathit{score}[z] \leftarrow \oplus^{post}(s^\mathit{pre}, n_z)$
20: $D_u.\mathit{predicted} \leftarrow \mathit{argtop}^{klocal}(\mathit{score}[z])$

$\triangleright$ Step 3: Compute recommendations

and $(Γ^\mathit{max}_{klocal})^2$ is defined in relation to $Γ_{k-local}$ as in (1).

The resulting GAS program is shown in Algorithm 2, and illustrated on a small example in Figure 4. The program comprises three GAS steps, each made of a gather (gather and sum functions), and an apply phase (apply function). We do no use any scatter phase. We employ the notation used in [25] where $D_u$, $D_v$, and $D_{(u,v)}$ are the program state and meta-data for vertices $u$, $v$ and edge $(u,v)$ respectively.

In the first step (lines 1-6, Fig. 4b), we construct a list of neighbors by collecting the ids of adjacent vertices. To limit memory overheads, we limit the size of the neighborhood set $D_u.Γ_u$ to the truncation threshold $thr^\Gamma$. This uses a uniform
Step 1

Snaple can be designed using distributed deployments. We present a detailed experimental evaluation of Snaple’s scoring framework and the dataset used in the evaluation.

Table 4: The datasets used in the evaluation

| dataset  | |V|   | |E|   | domain                      |
|----------|----|-----|----|----|----------------------------|
| gowalla  | [8] | 196,591 | 9.95M | social network |
| pokec    | [40] | 16M | 30.6M | social network |
| orkut    | [28] | 3M | 223M | social network |
| livejournal | [2] | 4.8M | 68.9M | co-authorship |
| twitter-rv | [18] | 41M | 1.4B | microblogging |


5.1 Experimental setup

We have implemented SNAPLE on top of GraphLab version 2.2 compiled using GCC 4.8 with C++11 compatibility enabled. We use the Warp engine offered by GraphLab with its default configuration. We run our experiments on the Grid5000 testbed (https://www.grid5000.fr) using two kind of computing nodes: type-I and type-II machines. Type-I nodes are equipped with 2 Intel Xeon L5420 (2.5 Ghz) processors, 4 cores per processor (8 cores per node), 32 GBytes of memory, and Gigabit Ethernet. Type-II nodes are more powerful machines designed for big-data workloads with 2 Intel Xeon E5-2660v2 (2.2 Ghz) processors per node, possessing each 10 cores (20 cores per node), 128 GBytes of memory and 10-Gigabit ethernet connection. We deploy our experiments on up to 32 type-I nodes (256 cores) and up to 8 type-II nodes (160 cores). The experiments are executed in type-I machines if not indicated otherwise. All the nodes have access to a shared network file system (NFS) where we store the graphs to be loaded.

5.2 Evaluation protocol

For our experiments, we use the set of publicly available datasets described in Table 4. They consist of a set of static directed graphs, except gowalla and orkut which are undirected. We transform these last two datasets into directed graphs by duplicating edges in both directions. These datasets provide a representative set of graphs from different domains and cover a broad range of sizes, from small instances with less than 1 million edges (gowalla) to large graphs with over 1 billion edges (twitter-rv).

In order to emulate the prediction process we follow a

random variable $\text{rand}()$ over $[0, 1]$ (line 3) to approximate the truncation under GAS constraints.

In the second step (lines 7-11, Fig. 4c), we use the truncated neighborhoods $D_u, \hat{\Gamma}_v$ to compute raw similarities between $u$ and each of its neighbors $v$ (line 8). At the end of this step, we only keep the top $k_{\text{local}}$ neighbors of $u$ (argtopk\text{local} operator, line 11), and store them into a dictionary $D_u.sims$ with their corresponding raw similarity $s_v$. When this second step terminates, the keys of $D_u.sims$ are the vertices of $\Gamma_{k_{\text{local}}} (u)$ as defined in (11).

In the final step (lines 12-20, Fig. 4d), we first compute path similarities (Sec. 3.1) using the combinator $\odot$ (line 15). These path similarities are limited to the $k_{\text{local}}$ 2-hop neighborhood of $u$, i.e. to paths $u \rightarrow v \rightarrow z$ such that $v \in D_u.sims.keys$ (line 13) and $z \in D_v.sims.keys$ (line 15). We then aggregate the path-similarities leading to the same candidate vertex $z$ as explained in Section 3.2 using the $\odot_{\text{pre}}$ (line 16) and $\odot_{\text{post}}$ operators (line 19). More precisely, $\gamma_1, \gamma_2, \gamma_3$ and $\Sigma_3$ are sets of triplets $(z, s_z, n_z) \in V \times \mathbb{R} \times \mathbb{N}$ that associate each candidate vertex $z$ to a similarity $s_z$ and a counter $n_z$ counts the number of paths over which $s_z$ has been accumulated. The function merge at line 16 performs a double fold (or reduce) operation on the vertices of $\gamma_1$ and $\gamma_2$ to compute the generalized sum and the count of equation (10), which are then used at line 19:

$$\text{merge}(\odot_{\text{pre}}^{\text{pre}} \gamma_1, \gamma_2) = \left\{ \left( z, \sum_{(x, s_x, n_x) \in \gamma_1 \cup \gamma_2}^{\text{pre}} s_x, \sum_{(x, s_x, n_x) \in \gamma_1 \cup \gamma_2} n_x \right) \mid z \in \gamma_1 V \cup \gamma_2 V \right\}$$

where $\gamma_1 V$ is the projection of $\gamma_1$ on its first component $V$. The program finally returns the top $k$ vertices with the best scores as predictions (line 20).

5. EVALUATION

We present a detailed experimental evaluation of SNAPLE implemented on top of the Graphlab distributed graph engine [11]. Graphlab implements the GAS paradigm over an asynchronous distributed shared memory abstraction, and is specifically designed for processing large graphs in distributed deployments.

We explore the space of possible link predictors that can be designed using SNAPLE’s scoring framework and the implementation presented in Algorithm 2. In particular, we evaluate how different configurations modify the quality of link predictions, and the total computing time.
similar approach to the one taken in [36]. We randomly remove one outgoing edge from each vertex with \( |\Gamma(u)| > 3 \). After the execution, we obtain \( k \) (with \( k = 5 \) fixed) predictions for each vertex. Experiments with SNAPle are parametrized by a score combination (taken from Table 3), a truncation threshold \( \text{thr}_T \), and a sampling parameter \( k_{\text{local}} \). Unless indicated otherwise \( \text{thr}_T = 200 \). We use Jaccard’s coefficient as raw similarity (function \( f \) in equation (6)). The linear combinator is configured with \( \alpha = 0.9 \), which was found to return the best predictions.

We evaluate approaches against two metrics: recall and execution time. Recall is the proportion of removed edges that are successfully returned by the algorithm. Execution time is measured from when the graph has been successfully loaded (as reported by GraphLab) until after all predictions have been computed. We ignore the time spent to load the graph from primary storage, as this time depends on the time is measured from when the graph has been successfully loaded (as reported by GraphLab) until after all predictions have been computed. We ignore the time spent to load the graph from primary storage, as this time depends on the primary storage, as this time depends on the

### 5.3 Comparison with a direct implementation

We first compare SNAPle against a direct implementation of Algorithm 1 on GraphLab using Jaccard’s coefficient. For a fair comparison, we limit the search of candidates to 2-hop neighborhoods, as in SNAPle. We call this implementation BASELINE. As discussed in Section 2.4, BASELINE needs to directly compute similarity between every pair of vertices that lie 2 hops away from each other. Because the GAS model only provides access to direct neighbors, BASELINE must propagate and store neighborhood information along every 2-hop path, resulting in substantial overheads.

We compare BASELINE against 12 configurations of SNAPle, by using three scoring configurations from Table 3 (linearSum, counter, and PPR), and by varying the truncation threshold \( \text{thr}_T \) and the sampling parameter \( k_{\text{local}} \) between \( \infty \) (in effect no truncation, resp. no sampling) and 20 (a low value to exacerbate the effect of each parameter). We execute BASELINE and the 12 SNAPle configurations on 4 type-II nodes (80 cores), and apply them to the datasets gowalla, pokec and livejournal. (orkut and twitter-rv cause BASELINE to fail by exhausting the available memory.) Table 5 reports the recall values and execution times we obtain, with the recall gains (resp. speedups) shown in brackets for SNAPle configurations, computed against BASELINE.

The table shows that even without truncation or sampling (\( \text{thr}_T = k_{\text{local}} = \infty \)), SNAPle clearly outperforms BASELINE both in terms of recall (which more than doubles on all datasets) and execution times, with speedups ranging from 1.6 to 4.6. Truncation (\( \text{thr}_T = 20 \)) brings a small improvement in speed-up, while causing recall to go down slightly, be it with or without \( k_{\text{local}} \). The sampling parameter \( k_{\text{local}} \) has the largest impact on the execution time by far, yielding speedups ranging from 15.8 (pokec) up to 109 on gowalla when applied alone (\( \text{thr}_T = \infty \), while having a minimal impact on recall. Adding truncation to \( k_{\text{local}} \) further improves execution times for the pokec (speedups ranging from 19 to 22.7) and livejournal (35.2 to 39.3) (last 3 lines of the table).

In terms of score configuration, linearSum, counter, and PPR tend to produce similar recall values, with a slight advantage for linearSum (best recalls are shown in bold). The differences in execution times between the three score configuration are within the experimental noise, and not significant.

This first experiment demonstrates the flexibility of SNAPle, which offers a large number of configurations. These results also illustrate the benefits of SNAPle in terms of recall and execution time, and highlight the importance of sampling to get high speedup while maintaining a good
recall. Although the effect of \( \tau \) is less pronounced than that of \( k_{\text{local}} \), truncation remains an important mechanism to deal with highly connected vertices. In Sections 5.5 and 5.7 we return to the impact of both parameters, with a more fine-grained sensitivity analysis of their effects. We otherwise fix \( \tau = 200 \) for our other experiments.

5.4 Scalability

We assess the scalability of SNAPLE by applying the \textit{linearSum} score to \textit{livejournal}, \textit{orkut} and \textit{twitter-rv} on varying numbers of \textit{type-I} and \textit{type-II} nodes. Figure 5 shows the results we obtain for two values of \( k_{\text{local}} \) (40 and 80). Other scoring configurations return near-identical results.

SNAPLE scales linearly with graph sizes (measured in edges), for both values of \( k_{\text{local}} \), and for \( k_{\text{local}} \in \{40, 80\} \). Missing points indicate configurations not fitting into memory.

![Graph execution times](image)

Figure 5: Snaple scales linearly with graph sizes (measured in edges). Execution times are reported for both \textit{type-I} and \textit{type-II} nodes, and for \( k_{\text{local}} \in \{40, 80\} \).

5.5 Impact of the truncation threshold \( \tau \)

The truncation of neighborhoods using \( \tau \) (Sec. 4) serves two purposes: to limit memory overheads, in particular in the case of densely connected components, and to improve execution times (particularly in conjunction with \( k_{\text{local}} \)).

By truncating neighborhoods, we might however lose relevant information, disturb the computation of raw similarities, and finally decrease the quality of link predictions. This effect is however limited to vertices whose neighborhood is larger than \( \tau \). To investigate this phenomenon, Figures 6b, 6c and 6d show the CDF of vertex out degrees across all three datasets, including very small minorities (around 1%) in the case of \textit{livejournal} and \textit{twitter-rv}.

![CDF of out degrees](image)

Figure 6: (a) Relative recall improvement for different \( \tau \) values against \( \tau = 10 \), with \textit{linearSum}. (b,c,d) The CDF of out degrees in each dataset shows how the recall stops improving when \( \tau \) goes beyond the out degree of 80% of the vertices in a dataset. Vertical bars indicate threshold values of 10, 20, 40, 80 and 100.

5.6 Impact of the vertex selection mechanism

We limit the number of paths to be explored by sampling the neighborhood of each node to a set with the \( k_{\text{local}} \) threshold.
most similar neighbors using the $\Gamma_{\text{max}}$ function (Step 2 of Algorithm 2). Discarding vertices reduces the computing time with the inconvenience of reducing the set of explored candidates. This has a direct impact on recall. Using $\Gamma_{\text{max}}$, we consider the most similar vertices to be good candidates for the sampling. In order to demonstrate that sampling using similarity improves the search of potential vertex candidates, and thus recall, we compare our proposed neighbors selection $\Gamma_{\text{max}}$ with $\Gamma_{\text{min}}$ and $\Gamma_{\text{nd}}$. As mentioned $\Gamma_{\text{max}}$ selects the $k_{\text{local}}$ neighbors with the largest similarity, the second one with the smallest similarity, and the last one selects a random neighbor.

Figure 7 shows the recall for various score methods in the livejournal dataset for different $k_{\text{local}}$ values. As $k_{\text{local}}$ grows the obtained recall converges for the three policies because we expand the search until we explore the same candidates in all cases. However, for small values of $k_{\text{local}}$, $\Gamma_{\text{max}}$ always gets larger recall than $\Gamma_{\text{min}}$ and $\Gamma_{\text{nd}}$. For $k_{\text{local}} = 5$, $\Gamma_{\text{max}}$ doubles $\Gamma_{\text{min}}$ recall and increases 50% compared to $\Gamma_{\text{nd}}$. This result indicates that using the similarity as a criterion to limit the amount of vertices to explore is particularly effective for small values of $k_{\text{local}}$. Reducing the number of vertices to explore reduces both the computation time and the amount of memory to use.

5.7 Impact of the sampling parameter $k_{\text{local}}$

The definition of $k_{\text{local}}$ puts an upper bound limit of $k_{\text{local}}^2$ candidate vertices to be scored. Reducing the search space reduces time and storage costs, but penalizes recall. However, different scores may get different recall values for the same $k_{\text{local}}$ value. We now analyze the impact of varying $k_{\text{local}}$ on the computing time and recall for the scoring methods proposed in Table 3.

Figure 8 shows the recall and computing time for livejournal and twitter-rv datasets. We reduce the presented figures to these datasets due to the lack of space. The Sum aggregator is exhaustive in the sense that it accumulates all pair-wise similarities, thus taking into account the popularity of a vertex $z$ (i.e., the number of paths connecting $u$ to $z$) in its final score. This behavior explains why the recall increases together with $k_{\text{local}}$. By contrast, Mean and Geom work differently. Mean improves the recall obtained with Sum for small values of $k_{\text{local}}$ (see linearMean). When $k_{\text{local}}$ increases the recall obtained with Mean goes down. We believe this may occur because Mean averages an increasing number of path-similarities for larger $k_{\text{local}}$ values, many of which might have low scores. Additionally, averaging eliminates information about the popularity of a vertex, contrarily to Sum. The Geom aggregator shows the same pattern in a stronger form, which is probably due to its sensitivity to non-similar vertices ($\text{sim}^*(u, z) = 0$).

5.8 Sensitivity to $k$ and to removed edges

For completeness, we analyze how recall evolves when we vary the number of answers Snaple returns. This is shown in Figure 9 for livejournal and pokec, and five score configurations, when $k$ takes the values 5, 10, 15, 20. In this range, the recall increases substantially with $k$. (The other scores based on the Mean and Geom aggregators follow a similar pattern.)

Similarly, we investigate the sensitivity of recall when we increase the number of edges that are removed per vertex. Removing more edges, we remove paths between vertices, which makes it more difficult for Snaple to find relevant vertices. As a result recall decreases (Figure 10). (If a vertex has fewer edges than the number to be removed, we removed all the edges except one.) The recall decreases proportionally to the number of removed edges. We observe similar patterns when using Mean and Geom aggregators.

5.9 Comparison to Cassovary

For completeness, we conclude our evaluation by comparing Snaple with a single-machine solution. This comparison serves two objectives: to assess the impact of Snaple when networking costs disappear, and to provide a reference point to gauge the benefits of distribution when processing very large graphs. We use Cassovary [42] for this comparison, a multithreaded in-memory graph library from Twitter. Cassovary is able to load relative large graphs and can traverse a graph fully allocated into main memory. It has been shown to be an efficient solution for computing random
It is also used in production by Twitter [12].

In a first attempt, we implemented the solution described by Algorithm 1 (with the 2-hop optimization). However, neither the recall nor computing time were competitive. We therefore moved on to a multithreaded version of the personalized page rank (PPR) [30] approximation based on random walks [19, 24]. It is also used in production by Twitter [12].

For each vertex we run random walks [19, 24]. It is also used in production by Twitter [12].

Figure 8: Comparison of computing time against recall for different scoring configurations on 256 cores. Each point corresponds to a different value of \( k_{local} = 5, 10, 20, 40, 80 \). The sum aggregator gets the highest recall, improving as \( k_{local} \) grows.

In a first attempt, we implemented the solution described by Algorithm 1 (with the 2-hop optimization). However, neither the recall nor computing time were competitive. We therefore moved on to a multithreaded version of the personalized page rank (PPR) [30] approximation based on random walks [37] to improve on these results. For each vertex we run \( w \) random walks of depth \( d \). \( d = 2 \) reaches the neighbors of a vertex; \( d = 3 \) its neighbors of neighbors and so on. Once the random walks terminate, the \( k \) most visited vertices not included into \( \Gamma(v) \) are returned as predictions.

Increasing \( w \) and \( d \), we force the algorithm to explore a larger number of vertices in a similar way we do when varying \( k_{local} \). We modify \( w \) and \( d \) configurations in order to find the largest recall in the shortest time.

Figure 11 compares the recall and computing time for Cassovary running on a type-II machine when varying \( w \) and \( d \) on the livejournal and twitter-rv datasets. We observe that increasing \( d \) does not necessarily improve recall, with \( d = 3 \) yielding recall values very close to that of larger values of \( d \). By contrast, larger values of \( w \) tend to yield better recall value, but they also significantly increase the computing time. In the case of twitter-rv, we run an extra configuration with \( d = 3 \) and \( w = 10000 \) getting 0.06 recall in 90 minutes. Unfortunately, we had to stop other configurations with larger \( d \) values and \( w = 10000 \) as they took too long to complete.

Snaple is designed to run on a distributed environment.
Table 6: Snaple also outperforms a state-of-the-art single-machine solution (results obtained on one type-II node)

<table>
<thead>
<tr>
<th>dataset</th>
<th>Cassovary recall (s)</th>
<th>Cassovary time(s)</th>
<th>Snaple recall (s)</th>
<th>Snaple time(s)</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>livejournal</td>
<td>0.24</td>
<td>93</td>
<td>0.30</td>
<td>45.8</td>
<td>2.03</td>
</tr>
<tr>
<td>twitter-rv</td>
<td>0.06</td>
<td>5420</td>
<td>0.08</td>
<td>600.7</td>
<td>9.02</td>
</tr>
</tbody>
</table>

Taking advantage of the speed of multiple simultaneous computing nodes. A comparison between Snaple on multiple machines and Cassovary on a single machine would therefore not be fair to Cassovary. For that reason, we compare the best results obtained in our previous analysis of Cassovary (best recall in the shortest time) with the results obtained running Snaple on a single machine. We use $k_{local} = 20$, which produces recall values close to that of Cassovary (slightly higher in fact). The results we obtain (Table 6) show that Snaple is faster than Cassovary (with speedups of 2.03 and 9.02) while increasing recall. This demonstrates that even on single machine deployments Snaple provides a competitive solution in terms of both prediction quality and execution time.

We can also use the results obtained with Cassovary to assess the benefits of distribution when processing very large graphs: the recall obtained by Cassovary on twitter-rv (0.06, Table 6) is obtained by Snaple in 177s (2min57s) when using linearSum with $k_{local} = 5$ on 256 type-I cores (32 type-I machines), as reported by Figure 8a (right-hand chart). This corresponds to a speedup of 30.62 against Cassovary, while only using 12.8 more cores.

6. RELATED WORK

The processing of large graphs on a single machine has been the subject of a number of works, ranging from cases in which the entire graph can fit into main memory [12], to more recent approaches exploiting secondary storage using specialized data structures, such as GraphChi [20] and X-Stream [33]. Although competitive and resource efficient, these approaches were not designed for distribution. As a result, they cannot scale beyond the limit of a single machine, and are unable to tap into the increased resources offered by a cluster, or a cloud infrastructure, which we can.

If we turn to distributed solutions, many works have proposed abstraction models to facilitate the distributed processing of large amounts of data, including graphs. The GAS engines we presented in Section 2 form one strand of this line of work. Related to the GAS paradigm, the Bulk Synchronous Parallel (or BSP) model [43] organizes a parallel computation into asynchronous supersteps that are then synchronized using barriers. Pregel [27] adapts the BSP model to graphs, and organizes computation around functions executing on vertices that exchange messages between each superstep.

The complexity of programming models such as BSP has led researchers to propose distributed shared memory solutions [6, 15]. However, the cost of remote memory accesses tends to make these solutions impracticable when results must be returned rapidly. An improvement has been proposed through PGAS (Parallel Global Address Space) [5] by reducing the number of remote memory accesses. The popularity of MapReduce solutions has also prompted the development of high-level languages such as Hive [41] or Pig [29] to encode distributed data computations. To unify these various models, GraphX [10] offers abstraction layers that combine different graph engines when performing a graph analysis, in an effort to shield developers from the details of each engine. In spite of their merits, none of these solutions considers link-prediction, and none addresses the problem of non-locality when traversing graphs [26].

7. CONCLUSION

In this paper we have presented the design, implementation and evaluation of Snaple, a highly-scalable approach to the link-prediction problem optimized for the gather-apply-scatter (GAS) model of distributed graph engines. We have provided an exhaustive evaluation of our prototype in a cluster using a representative array of large publicly available datasets. Snaple is able to compute the predictions of a graph containing 1.4 billion edges in less than 3 minutes when other naive GraphLab solutions fail due to resource exhaustion. Additionally, we demonstrate that Snaple has an over-linear speedup of 30 when compared with a state-of-the-art non-distributed solution, while improving prediction quality.

Our work opens several exciting paths for future research. One such path involves the extension of Snaple to supervised link-prediction strategies, which may improve recall while taking advantage of distributed computing. We also would like to port Snaple to other distributed graph processing platforms such as Giraph [1], Bagel [3] or Stinger [9] to provide more comparison points between these platforms. Finally, we plan to perform a more in-depth analysis of the data structures used in our prototype to understand how individual operations can be further improved, and help reduce latency.

Acknowledgment

Experiments presented in this paper were carried out using the Grid’5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

8. REFERENCES
