Efficient Batch One-Hop Personalized PageRanks

Siqiang Luo
University of Hong Kong
sqluo@cs.hku.hk

Xiaokui Xiao
National University of Singapore
xkxiao@nus.edu.sg

Wenqing Lin
Tencent
edwlin@tencent.com

Ben Kao
University of Hong Kong
kao@cs.hku.hk

***Abstract*—Personalized PageRank (PPR) is a classic measure of the relevance among different nodes in a graph. Existing work on PPR has mainly focused on three general types of queries, namely, single-pair PPR, single-source PPR, and all-pair PPR. However, there are applications that rely on a new query type (referred to as *batch one-hop PPR*), which takes as input a set S of source nodes and, for each node $s \in S$ and each of s's neighbor v, asks for the PPR value of v with respect to s. None of the existing PPR algorithms is able to efficiently process batch one-hop queries, due to the inherent differences between batch one-hop PPR and the three general query types. To address the limitations of existing algorithms, this paper presents *Baton*, an algorithm for batch one-hop PPR that offers strong practical efficiency.**

I. Introduction

Given two nodes s and t in a graph G, the *Personalized PageRank (PPR)* of t with respect to s, denoted as $\pi(s,t)$, is defined as the probability that a random walk (with decay) from s would terminate at t. The importance of PPR has motivated numerous solutions [4], [7], [10], [11], [8] that aim to improve the efficiency of PPR computation. Existing solutions mainly address three types of PPR queries: 1) *single-pair* PPR, which returns $\pi(s,t)$ for a given pair (s,t); 2) *single-source* PPR, which returns $\pi(s,v)$ for a given s and every node v in G; 3) *all-pair* PPR, which returns $\pi(u,v)$ for all possible node pairs (u,v).

Although these generic query types cover a number of applications (e.g., [5], [6], [9]), we observe that there is often a need for more specialized form of PPR queries. In particular, we consider a problem that we encounter in Tencent's massive online gaming platform with a social network G of billions of users. The platform has a PPR-based mechanism that aims to attract *inactive* users back to the platform, and it works as follows. First, for each inactive user s, the platform inspects her friends in the social network, and identifies the ones who are active and have large PPR values with respect to s. Then, the platform asks each v of those friends to send a message to s to invite her back, and gives v a reward if s returns to the platform upon receiving the message. A/B tests show that this PPR-based mechanism is much more effective than other mechanisms considered. Nonetheless, the computation of PPR poses a significant challenge for deploying the PPR-based mechanism in Tencent, as the number of inactive users can be up to billions. That is, given a large subset S of the nodes in G, we need an efficient method to compute, for each node $s \in S$, the PPR values of s's neighbors with respect to s. We refer to this type of queries as *batch one-hop PPR queries*.

A naive solution to process batch one-hop PPR is to answer the query using existing algorithms for single-pair, single-source, or all-pair PPR queries; nevertheless, this incurs tremendous computation overheads. In particular, if we are to answer a batch one-hop query using a single-source algorithm, then we need to invoke the algorithm once for each node in S. Assuming that $|S| = 10^8$ and that each invocation of the algorithm requires 100 seconds (which is typical for the state of the art [11]), the total processing cost would be 10^{10} seconds ($\approx$ 317 years), which is prohibitive. Similarly, answering the query using a single-pair PPR algorithm would result in efficiency issues, because (i) we need to apply the algorithm once for each edge adjacent to the nodes in S, and (ii) the number of such edges is often two orders of magnitude larger than $|S|$. All-pair PPR algorithms are inapplicable, either, as their $O(n^2)$ space overheads restrict their application to small graphs only.

Contributions. This paper presents a comprehensive study on batch one-hop PPR queries, and proposes Baton (**Bat**ch **O**ne-Hop Perso**n**alized PageRanks), an algorithm that offers strong practical efficiency. Our experimental evaluation shows that Baton outperforms the state of the arts by several orders of magnitude in terms of running time.

II. Preliminaries

A. Problem Definition

Let $G(V,E)$ be a directed graph with node set V and edge set E. Given a source node $s \in V$ and a decay factor α, a random walk from s is a traversal of G starting from s, such that at each step of the traversal, it terminates with α probability and, with the other $1-\alpha$ probability, moves to a randomly selected out-neighbor of the current node. For any node t, the *Personalized PageRank (PPR)* [9] of t with respect to s, denoted as $\pi(s,t)$, is defined as the probability that a random walk from s stops at t. We aim to answer *batch one-hop PPR queries* with accuracy guarantees, defined as follows.

Definition 1 (Approximate Batch One-Hop PPR Queries): Given a set S of nodes in a graph G, a threshold δ, an error bound ϵ, and a failure probability p_f, an approximate batch one-hop PPR query returns an estimated PPR $\hat{\pi}(s,v)$ for every node pair (s,v) such that $s \in S$ and v is an out-neighbor of S, such that for all $\pi(s,v) \geq \delta$,

$$|\pi(s,v) - \hat{\pi}(s,v)| \leq \epsilon \cdot \pi(s,v) \tag{1}$$

holds with a probability at least $1 - p_f$. □

2375-026X/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDE.2019.00142

Algorithm 1: Forward-Push(G, s, r_{max}, α)

```
    Input: Graph $G$, source node $s$, residue threshold $r_{max}$,
           probability $\alpha$
    Output: $\pi^\circ(s,u)$, $r(s,u)$ for all $u \in V$
1   for $u \in V$ do
2       $r(s,u) = 0$, $\pi^\circ(s,u) = 0$, $d(u) =$ out degree of $u$
3   $r(s,s) = 1$
4   while exists $u \in V$ such that $r(s,u) > r_{max} \cdot d(u)$ do
5       Push-Step($G$, $s$, $\alpha$, $u$)
```

Algorithm 2: Push-Step(G, s, α, u)

```
1   for each $v$ that is an out-neighbor of $u$ do
2       $r(s,v) = r(s,v) + (1-\alpha) \cdot \frac{r(s,u)}{d(u)}$
3   $\pi^\circ(s,u) = \pi^\circ(s,u) + \alpha \cdot r(s,u)$
4   $r(s,u) = 0$
```

B. Main Competitors

Monte-Carlo (MC). The MC method [4] is a classical solution for PPR estimation. To achieve the accuracy guarantee in Equation 1, it generates $\omega = \Omega\left(\frac{\log(1/p_f)}{\epsilon^2\delta}\right)$ random walks starting at s to estimate $\pi(s,t)$ for every node t. If ω' of them terminate at t, then $\frac{\omega'}{\omega}$ is an unbiased estimate of $\pi(s,t)$.

Forward Push. Forward push [3] is a method for answering single-source PPR queries (see Algorithm 1). It maintains, for each node $u \in V$, a *reserve* $\pi^\circ(s,u)$ and a *residue* $r(s,u)$, which are dynamically updated by a propagation process from the source node s. Initially, all reserves and residues are set to 0, except that the residue of s is set to 1. The propagation is then repeatedly conducted based on Algorithm 1. In brief, conducting a forward push on node u transfers α portion of its residue to its reserve, while the remaining $(1-\alpha)$ portion is equally distributed to the out-neighbors of u. It can be shown that when the residue threshold r_{max} is set close to 0, the final reserves are close to the actual PPR scores.

BiPPR and HubPPR. BiPPR [7] is a method for single-pair PPR queries that improves over MC and forward push. Given a node pair (s,t), BiPPR conducts a number of random walks from s as well as a *reverse push* [2] from t, and then combines the information obtained to derive an estimation of $\pi(s,t)$. The reverse push algorithm is similar in spirit to the forward push method, except that (i) it follows the incoming edges of each node instead of the outgoing edges, and (ii) it derives the residue and reserve of each node in a different manner. It is shown in [7] that for randomly chosen t, BiPPR requires $O\left(\sqrt{\frac{m\log(1/p_f)}{n\epsilon^2\delta}}\right)$ expected time to achieve the accuracy guarantee in Equation 1, which is a significant improvement over MC and forward push. HubPPR [10] is an enhancement of BiPPR that it (i) improves query efficiency with indexing and (ii) retains the theoretical guarantees of BiPPR.

FORA. FORA [11] is the state-of-the-art method for single-source PPR queries, and it is based on a combination of MC and forward push. Specifically, it first conducts a forward push with threshold r_{max} from the source node s, and then performs random walks from each node v, such that the number of random walks from v is proportional to its residue. It is proved in [11] that, for each node $u \in V$, the estimate $\hat{\pi}(s,u) = \pi^\circ(s,u) + c(u)/K$ is an unbiased estimate of $\pi(s,u)$, where $\pi^\circ(s,u)$ is the reserve of u, K is the total number of random walks that have been performed, and $c(u)$ is the number of random walks that end at u. It is also shown that, by setting $K = O\left(r_{sum} \cdot \frac{(2\epsilon/3+2)\log(2/p_f)}{\epsilon^2\delta}\right)$, FORA achieves the accuracy guarantee in Equation 1, where r_{sum} is the sum of residues of nodes when the forward push terminates.

The key of FORA is to determine a good threshold r_{max} to balance the costs of the forward push phase and the random walk phase. Wang et al. [11] suggest setting $r_{max} = O\left(\frac{\epsilon}{\sqrt{m}} \cdot \sqrt{\frac{\delta}{(2\epsilon/3+2)\log(2/p_f)}}\right)$, so that the total cost of forward push and random walks is optimized as $O\left(\sqrt{\frac{(2\epsilon/3+2)m\log(2/p_f)}{\epsilon^2\delta}}\right)$. In addition, Wang et al. [11] also propose an indexed version of FORA, referred to as FORA+, that offers high query efficiency at the costs of space and preprocessing.

Adaptation to Batch One-Hop PPR. MC and FORA can be adopted to answer batch one-hop queries by performing one single-source query for each node s in S. Meanwhile, if we are to apply BiPPR and HubPPR to process batch one-hop queries, then we need to perform one single-pair query for each node pair (s,v), such that $s \in S$ and v is an out-neighbor of s. As we demonstrate in our experiments (see in Section IV), such adoptions result in inferior efficiency.

III. Our Solution

A. Lower Bound for One-Hop PPR

Let (s,v) be any node pair in G, and suppose that we are to estimate $\pi(s,v)$ with ϵ relative error. Intuitively, the estimation is more difficult when $\pi(s,v)$ is small, since the margin of error decreases with $\pi(s,v)$. (This also explains why the time complexities of MC, BiPPR, HubPPR, and FORA are all inverse proportional to the PPR threshold δ.) On the other hand, if we know in advance that $\pi(s,v)$ is large, then we could be less stringent in estimating $\pi(s,v)$, as there is more room for error. This motivates us to derive a lower bound for one-hop PPR values, so as to guide our algorithm for batch one-hop PPR. In particular, we have the following lemma [1].

Lemma 1: For any node s and any out-neighbor t of s, we have $\pi(s,t) \geq \frac{\alpha(1-\alpha)}{d(s)}$, where $d(s)$ is the out-degree of s.

The above result can be exploited to reduce the overhead of batch one-hop PPR queries. For example, consider the FORA algorithm (discussed in Section II-B), which answers any single-source PPR query from a node s in $O\left(\sqrt{\frac{m\log(1/p_f)}{\epsilon^2\delta}}\right)$

[1]All the proofs of lemmas can be found in [1].

expected time, and ensures ϵ relative error for any $\pi(s,v) \geq \delta$. Applying FORA to answer a batch one-hop query would require one single-source query for each node $s \in S$, leading to a total expected cost of $O\left(|S|\sqrt{\frac{m\log(1/p_f)}{\epsilon^2\delta}}\right)$.

As mentioned in Section II, δ is typically set to $O(1/n)$, which could be much smaller than $\alpha(1-\alpha)/d(s)$. Therefore, if we are to invoke FORA for a batch one-hop query, we can set $\delta = \alpha(1-\alpha)/d(s)$ instead. By Lemma 1, FORA would still ensure ϵ relative error in the estimation of $\pi(s,v)$, as long as v is an out-neighbor of s. As such, the expected cost of using FORA to process the query is

$$O\left(\sum_{s\in S}\sqrt{\frac{m\log(1/p_f)}{\epsilon^2(\alpha(1-\alpha)/d(s))}}\right) = O\left(\sqrt{\frac{m\log(1/p_f)}{\epsilon^2}}\sum_{s\in S}\sqrt{d(s)}\right)$$

In contrast, setting $\delta = O(1/n)$ would result in a total expected cost of $O\left(\sqrt{\frac{m\log(1/p_f)}{\epsilon^2}}\cdot|S|\sqrt{n}\right)$, which is inferior since $|S|\sqrt{n} > \sum_{s\in S}\sqrt{d(s)}$ holds. Similarly, we can incorporate the lower bound in Lemma 1 into BiPPR and HubPPR, so as to reduce their expected time complexities for batch one-hop queries.

B. The Baton Method

To better utilize the lower bound in Lemma 1, we present the *Baton* method shown in Algorithm 3 for batch one-hop PPR queries. At the first glance, Baton may seem similar to FORA as they both perform forward push from each node $s \in S$, followed by generating random walks from the nodes with non-zero residues. There is one crucial difference, however: Baton's forward push phase performs a *push step* on a node u whenever $r(s,u) > \frac{d(u)}{\alpha\cdot K(s)}$ (2), where $K(s) = \frac{(\frac{2}{3}\epsilon+2)d(s)\log(2/p_f)}{\epsilon^2\alpha(1-\alpha)}$ is a constant that increases with the out-degree $d(s)$ of s (see Lines 2-4 in Algorithm 3); in contrast, FORA's forward push phase applies a push step on u whenever $r(s,u) > d(u)\cdot r_{max}$ (3), where $r_{max} = O\left(\frac{\epsilon}{\sqrt{m}}\cdot\sqrt{\frac{\delta}{(2\epsilon/3+2)\log(2/p_f)}}\right)$ is a constant independent of s. In other words, Baton is more likely to "push" when $d(s)$ is large, whereas FORA does not consider $d(s)$ when deciding whether a push step is needed. In what follows, we will explain (i) the rationale between these two design choices, (ii) why our design is non-trivial with respect to FORA, and (iii) how our design leads to significantly improved performance.

First, when the "push condition" in Equation 3 is adopted, the forward push method (i.e., Algorithm 1) runs in $O(1/r_{max})$ time [3]. FORA relies on this result to bound the cost of its forward push phase [11], and hence, it also adopts Equation 3. As such, changing the push condition from Equation 3 to Equation 2 is non-trivial, as it invalidates the time complexity analysis in [11], and requires new analytical results to be derived for the revised forward push method.

Second, the reason that Baton uses the push condition in Equation 2 is that it helps Baton achieve improved asymptotic performance by striking a better balance between forward push and random walks. To explain, suppose that we encounter, in the forward push phase, a node u with reserve $\pi^\circ(s,u)$ and residue $r(s,u)$. If we choose not to perform a push step on u, then according to Lines 8-12 in Algorithm 3, the random walk phase would need to generate $r(s,u)\cdot K(s)$ random walks from u. On the other hand, if we apply a push step on u, then u's out-neighbor's total residue is increased by $(1-\alpha)\cdot r(s,u)$, and then u's residue is reset to 0; in that case, the random walk phase needs to generate $(1-\alpha)\cdot r(s,u)\cdot K(s)$ random walks from u's out-neighbors, but does not require any random walk from u. Therefore, performing the push step on u reduces the number of random walks required by $\alpha\cdot r(s,u)\cdot K(s)$, at the cost of $O(d(u))$ computation (since each of u's out-neighbor needs to be visited). This explains why Baton's push condition is $r(s,u) > d(u)/(\alpha\cdot K(s))$: it ensures that $d(u) < \alpha\cdot r(s,u)\cdot K(s)$, which roughly indicates that a push step on u could reduce the total computation cost of the forward push and random walk phases.

Algorithm 3: Baton(G, S, ϵ, p_f, α)

```
Input: Graph $G$, source node set $S$, PPR relative
       accuracy guarantee $\epsilon$, failure probability $p_f$,
       probability $\alpha$
Output: PPR estimate $\hat{\pi}(s,u)$, for all $s \in S$, $u \in N(s)$
1  for $s \in S$ do
2      $K(s) = \frac{(\frac{2}{3}\epsilon+2)d(s)\log(2/p_f)}{\epsilon^2\alpha(1-\alpha)}$
3      while exists $u$ such that $r(s,u) > \frac{d(u)}{\alpha\cdot K(s)}$ do
4          Push-Step($G$, $s$, $\alpha$, $u$) (by Algorithm 2)
5      for $t \in N_{out}(s)$ do
6          $\hat{\pi}(s,t) = \pi^\circ(s,t)$
7      for $v \in V$ and $r(s,v) > 0$ do
8          for $i = 1$ to $(r(s,v)\cdot K(s))$ do
9              Conduct a random walk from $v$
10             if the random walk terminates at $t$ then
11                 if $t \in N_{out}(s)$ then
12                     $\hat{\pi}(s,t) = \hat{\pi}(s,t) + \frac{1}{K(s)}$
```

IV. Experiments

Settings. Our tested graphs are shown in Table I. We compare Baton with MC [4], HubPPR [10], FORA+ [11], and the respective optimized versions of the state-of-the-art algorithms HubPPR-OPT and FORA-OPT using the tightened accuracy bounds (Lemma 1). FORA-OPT is the improved method we described in Section III-A. (we name the method FORA-OPT to distinguish it with the original FORA.) Details about HubPPR-OPT can be found in [1]. The implementations of HubPPR [10] and FORA+ [11] are obtained from their respective authors. All the methods are implemented by C++. For each dataset, we choose 1000 source nodes uniformly at random to compute the 1000 one-hop PPR queries, whose running times are averaged and reported. Following [7], [10], [11], we set $\delta = 1/n$, $p_f = 1/n$, and $\epsilon = 0.5$ by default. Our

TABLE I: Datasets. ($K = 10^3$, $M = 10^6$, $B = 10^9$)

Name	n	m	Type	Additional Info.
DBLP	613.6K	2.0M	undirected	dblp.com
Web-St	281.9K	2.3M	directed	stanford.edu
Pokec	1.6M	30.6M	directed	pokec.azet.sk
LJ	4.8M	69.0M	directed	livejournal.com
Orkut	3.1M	117.2M	undirected	orkut.com
Twitter	41.7M	1.5B	directed	twitter.com

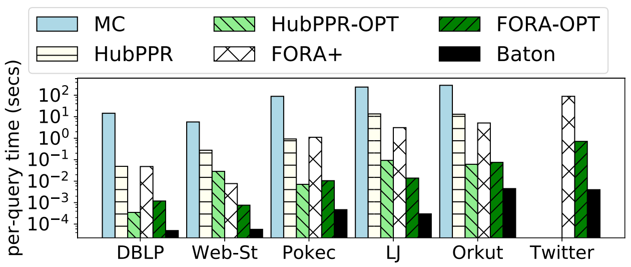

Fig. 1: Efficiency.

experiments are conducted on a Linux machine with an Intel 2.6GHz CPU and 64GB memory.

Performance. Figure 1 (a) shows the per-query efficiency of the methods. We omit the running times of HubPPR, HubPPR-OPT and MC for the largest Twitter dataset, because 1) MC fails to finish within 500 seconds per query; 2) We are not able to build the index of HubPPR/HubPPR-OPT for the billion-edge graph, due to the excessively large memory required.

As expected, the classic algorithm MC runs relatively slow, and if S is a large subset of V, it would fail to handle a moderate sized graph Pokec. For example, let us set $|S| = 1M$. Then, if to estimate the overall cost based on the scaled average query time, MC takes 89s×1M=2.8 years to compute the batch one-hop PPRs for Pokec, which is not satisfactory. The state-of-the-art algorithms, i.e., FORA and HubPPR, significantly outperform MC. However, they are still much slower compared with their respective optimized algorithms, i.e., FORA-OPT and HubPPR-OPT. The improvement of FORA-OPT and HubPPR-OPT is expected because the tightened bound indicated in Lemma 1 allows the number of random walks to be significantly reduced, while still maintaining the worst-case accuracy guarantee.

Baton consistently outperforms the other algorithms. In particular, Baton is around 3 orders of magnitude faster than FORA and HubPPR. Even compared with FORA-OPT and HubPPR-OPT, Baton is still around one order of magnitude faster. That means, among all the methods we compare, Baton is the most suitable to process batch one-hop queries, due to its significant improvement in efficiency.

As we analyzed in Section III-A, Baton employs an optimized mechanism to minimize the overall cost. To illustrate, we summarize the average costs of forward push phase and random walk phase for FORA-OPT and Baton on the representative datasets in Table II. The push cost is defined by the number of executing Line 2 in Algorithm 2. For example, when a push-step (Algorithm 2) is performed on node u, Line 2 is executed $d(u)$ times, and therefore $d(u)$ reflects the cost of conducting a push step. The cost of random walks, as we mentioned in Section III-B, is reflected by the number of random walks. From Table II we observe that FORA-OPT is *overly using push*. One can refer to the numbers of Baton for LJ dataset, which indicates that by conducting pushes at a cost of 1064, the remaining workload of random walks becomes 5918. However, FORA-OPT uses 166220/1064=156 times more push costs than Baton, only to reduce the workload of random walks by $(5918 - 3134)/5918 = 47\%$. This over-use of push renders FORA-OPT being significantly outperformed by Baton. In particular, as shown in Figure 1, Baton is 25.4 times faster than FORA-OPT in DBLP, and 67.5 times faster in LJ. The superiority of Baton over FORA-OPT agrees well with our analysis in Section III-A.

TABLE II: Statistics of two phases in FORA-OPT and Baton.

Dataset (method)	Push cost	Random walk cost	Total
DBLP (FORA-OPT)	29,956	746	30,702
DBLP (Baton)	607	1,864	2,471
LJ (FORA-OPT)	166,220	3,134	169,354
LJ (Baton)	1,064	5,918	6,982

V. Conclusion

We conducted a comprehensive study on the batch one-hop PPR queries.We propose Baton, an adaptive mechanism that answers the batch one-hop PPR queries cost-effectively. Baton incorporates various optimizations, resulting in a method that is orders of magnitude faster than the state of the arts.

Acknowledgments

This work is supported in part by grant MOE2015-T2-2-069 from MOE Singapore.

References

[1] "https://sites.google.com/view/baton-ppr," *Technical Report*, 2018.
[2] R. Andersen, C. Borgs, J. Chayes, J. Hopcraft, V. Mirrokni, and S.-H. Teng, "Local computation of pagerank contributions," *Algorithms and Models for the Web-Graph*, pp. 150–165, 2007.
[3] R. Andersen, F. Chung, and K. Lang, "Local graph partitioning using pagerank vectors," in *FOCS*, 2006, pp. 475–486.
[4] D. Fogaras, B. Rácz, K. Csalogány, and T. Sarlós, "Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments," *Internet Mathematics*, vol. 2, no. 3, pp. 333–358, 2005.
[5] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh, "Wtf: The who to follow service at twitter," in *WWW*, 2013, pp. 505–514.
[6] D. C. Liu, S. Rogers, R. Shiau, D. Kislyuk, K. C. Ma, Z. Zhong, J. Liu, and Y. Jing, "Related pins at pinterest: The evolution of a real-world recommender system," in *WWW (Companion)*, 2017, pp. 583–592.
[7] P. Lofgren, S. Banerjee, and A. Goel, "Personalized pagerank estimation and search: A bidirectional approach," in *WSDM*, 2016, pp. 163–172.
[8] T. Maehara, T. Akiba, Y. Iwata, and K.-i. Kawarabayashi, "Computing personalized pagerank quickly by exploiting graph structures," *VLDB*, vol. 7, no. 12, pp. 1023–1034, 2014.
[9] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.
[10] S. Wang, Y. Tang, X. Xiao, Y. Yang, and Z. Li, "HubPPR: effective indexing for approximate personalized pagerank," *VLDB*, vol. 10, no. 3, pp. 205–216, 2016.
[11] S. Wang, R. Yang, X. Xiao, Z. Wei, and Y. Yang, "FORA: Simple and effective approximate single-source personalized pagerank," in *KDD*, 2017, pp. 505–514.