BATON: Batch One-Hop Personalized PageRanks with Efficiency and Accuracy

Siqiang Luo, Xiaokui Xiao, Wenqing Lin, Ben Kao

Abstract—Personalized PageRank (PPR) is a classic measure of the relevance among different nodes in a graph, and has been applied in numerous systems, such as Twitter's Who-To-Follow and Pinterest's Related Pins. 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, we observe that 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 both strong theoretical guarantees and practical efficiency. Baton leverages the characteristics of one-hop PPR to avoid unnecessary computation, and it incorporates advanced mechanisms to improve the cost-effectiveness of PPR derivations. Extensive experiments on benchmark datasets show that Baton is up to 3 orders of magnitude faster than the state of the art, while offering the same accuracy.

Index Terms—Personalized PageRanks, Query Performance, Graph Algorithms

1 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. PPR is a classic approach to measure the relevance of t with respect to s, and has been adopted in numerous systems. For example, Twitter utilizes PPR to recommend users to other users [11], and Pinterest applies PPR for content recommendations [13].

The importance of PPR has motivated numerous solutions [8], [2], [1], [15], [30], [31], [9], [19], [26], [33] that aim to improve the efficiency of PPR computation. Existing solutions mainly address three types of PPR queries:

- *single-pair* PPR, which returns $\pi(s,t)$ for a given pair (s,t);
- *single-source* PPR, which returns $\pi(s,v)$ for a given s and every node v in G;
- *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., [11], [13], [12], [21]), 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 (Detailed statistics can be referred to Section 4.3). 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 [31]), the total processing cost would be 10^{10} seconds ($\approx$ 317 years), which is prohibitive even if we can distribute the computation to a large number of machines. 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

- *Siqiang Luo and Ben Kao are with the Department of Computer Science, the University of Hong Kong.*
  e-mail: {sqluo, kao}@cs.hku.hk
- *Xiaokui Xiao is with the School of Computing, National University of Singapore, Singapore.*
  e-mail: xkxiao@nus.edu.sg
- *Wenqing Lin is with Tencent, Shenzhen, China.*
  e-mail: edwlin@tencent.com

$O(n^2)$ 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[1], an algorithm that offers both strong theoretical guarantees and practical efficiency. Given a set S of source nodes and parameters $p_f, \epsilon \in (0,1)$, Baton returns one-hop PPR for each node $s \in S$, such that with at least $1 - p_f$ probability, all PPR values returned have at most ϵ relative error. This accuracy guarantee matches those provided by the most advanced methods for single-pair and single-source PPR. In addition, Baton runs in $O\left(\sum_{s \in S} \frac{d(s)\log(1/p_f)}{\epsilon^2}\right)$ expected time (where $d(s)$ denotes the degree of s), which is linear to the total number of edges adjacent to the nodes in S.

Baton is built upon FORA [31] (i.e., the state-of-the-art approach for single-source PPR), but incorporates three new techniques that yield significantly improved performance for batch one-hop queries. The first technique is based on a careful analysis of the characteristics of one-hop PPR, and it enables us to tighten the accuracy bounds of Baton without extra computation overheads. The second technique is an adaptive graph traversal mechanism that considerably improves the cost-effectiveness of Baton in deriving one-hop PPR values, and we prove that the mechanism is a local minimum among a general class of similar approaches. The third technique is based on vertex covers, and it significantly reduces Baton's computation overheads on undirected graphs.

We experimentally evaluate Baton on 10 datasets (including various Tencent datasets and public datasets) with up to 74.3 million nodes and 1.5 billion edges. Our results show that Baton outperforms the state of the art by several orders of magnitude in terms of running time, while offering the same degree of accuracy. In particular, on a billion-edge Tencent game-user social network, Baton requires only 2.48 milliseconds on average to process each node $s \in S$ in a batch one-hop query.

A preliminary version of this work appears in [18]. This paper is a significant extension of [18] in the following aspects. First, we give a much deeper theoretical analysis on the Baton algorithm. We show the complexity of Baton (Lemma 3), and prove that this is close to optimal. We introduce the concept of *generic-baton* that generalizes FORA and Baton. We show that the Baton algorithm is a local optimum (Lemma 4), which gives evidence why Baton could be better than FORA in batch one-hop queries. Second, we present new techniques. We discuss the random walk reusing technique in Section 3.3, and propose new optimization approach for undirected graphs in Section 3.4. Third, we extend the Baton algorithm to run on multi-core servers (Section 3.5). Finally, we give abundant experiments, including tests on new datasets, new experiments to show that Baton offers strong NDCG accuracy in approximating PageRank values, as well as the A/B test performed on the Tencent platform (in Section 4.3) to show the superiority of the one-hop PPR based method (e.g., Baton) over four other approaches in social network applications.

1. **Bat**ch **O**ne-Hop Perso**n**alized PageRanks

TABLE 1: Frequently used notations

Notation	Description
$G(V,E)$	Input graph
n	Number of nodes in G
m	Number of edges in G
$\pi(s,u)$	PPR of node u with respect to node s
$\bar{d}$	Average node degree in G
$d(s)$	Out-degree of node s
ϵ	PPR relative accuracy guarantee
p_f	Failure probability
δ	Threshold of PPR values
$N_{out}(v)$	The set of out-neighbors of v
C	Vertex cover of G
$r(s,u)$	Residue of node u, with respect to source node s
$\pi^\circ(s,u)$	Reserve of node u, with respect to source node s
$\hat{\pi}(s,u)$	Estimate of $\pi(s,u)$

The remainder of the paper is organized as follows. Section 2 defines the problem and discusses some related work. Section 3 explains the details of Baton. Section 4 presents experimental results. Finally, Section 5 concludes the paper.

2 Preliminaries

In this section we present the problem definition of the batch one-hop queries and the related works.

2.1 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)* [21] 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 $1 - p_f$. □

Our accuracy guarantee (i.e., ensuring ϵ relative error whenever $\pi(s,v) \geq \delta$) is consistent with those of the state-of-the-art solutions for single-pair and single-source PPR [16], [15], [30], [31]. As suggested in [16], [15], [30], [31], δ is typically set to $\delta = O(1/n)$, because for any fixed s, the average value of $\pi(s,v)$ over all possible $v \in V$ is $1/n$. Table 1 shows the notations that we frequently use in this paper.

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

```
Input: Graph $G$, probability $\alpha$, source node $s$,
       residue threshold $r_{max}$
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 $r(s,u) = 0$
```

2.2 Main Competitors

Monte-Carlo. The Monte-Carlo (MC) method [8] is a simple and classical solution for PPR estimation. It generates a number of random walks starting at s to estimate $\pi(s,t)$ for every node t. In particular, if ω random walks are generated and ω' of them terminate at t, then $\frac{\omega'}{\omega}$ is an unbiased estimate of $\pi(s,t)$. It has been shown in [8] that, to achieve the accuracy guarantee in Equation 1, we should have $\omega = \Omega\left(\frac{\log(1/p_f)}{\epsilon^2\delta}\right)$.

Forward Push. Forward push [2] is a method for answering single-source PPR queries (see Algorithm 1 for a pseudo-code). 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 forward push runs in $O(1/r_{max})$ time, and that when the residue threshold r_{max} is set close to 0, the final reserves are close to the actual PPR scores. However, as pointed out in [31], forward push has one main deficiency: it can either compute the exact single-source PPR results at a high cost, or terminate early but with no guarantee on the result quality.

BiPPR and HubPPR. BiPPR [15] 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* [1] 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 [15] 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 [30] is an enhancement of BiPPR that it (i) improves query efficiency with indexing and (ii) retains the theoretical guarantees of BiPPR.

FORA. FORA [31] 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 [31] 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 should be performed if only using MC, 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. [31] 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. [31] 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 are single-source PPR algorithms, and hence, they can be adopted to answer batch one-hop queries by performing one single-source query for each nodes in S. Meanwhile, both BiPPR and HubPPR are single-pair PPR methods; if we are to apply them 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 (Section 4), such adoptions result in inferior efficiency, due to the inherent differences between batch one-hop and single-pair/-source queries.

2.3 Other Related Work

Besides the algorithms mentioned in Section 2.2, a lot of research effort has been spent on answering single-pair, single-source, or all-pair PPR queries. However, the existing studies are either difficult to be applied to the batch one-hop PPR queries due to prohibitive complexities, or not as efficient as HubPPR or FORA.

Single-Source PPR. Many studies on single-source PPR queries leverage the matrix-form of the PPR as $\pi_s = \alpha \cdot e_s + (1-\alpha) \cdot \pi_s \cdot D^{-1}A$, where $A \in \{0,1\}^{n\times n}$ is the adjacency matrix for the input graph G, and $D \in R^{n\times n}$ is a diagonal matrix whose i-th diagonal element equals to the out-degree of v_i. It can be shown that the solution vector π_s of the matrix-form has its i-th element equal to $\pi(s,v_i)$. Hence, a typical matrix-based solution to evaluating π_s is to conduct power iteration: 1) start with an initial guess of π_s, e.g., $\{\frac{1}{n}\}^{1\times n}$; 2) iteratively refine the guess of π_s

with matrix-form equation. Simply using the power iteration is costly due to many rounds of matrix-multiplications required. Therefore, there are a number of studies [9], [19], [26], [33], [24] that leverage this basic idea of power iteration, but with various improvements. Among them, the BEAR algorithm proposed by Shin et al. [26] is the state-of-the-art algorithm. BEAR reorders the adjacency matrix to obtain several sparse submatrices, which are less costly to conduct invert operations. The sub-matrices are then indexed, and used to answer PPR queries. In a recent work [30], however, BEAR is shown to be inferior to HubPPR in terms of query efficiency and accuracy.

Single-Pair PPR. There are a number of earlier studies that can be applied to improve the efficiency of evaluating single-pair PPR queries, based on the sampling (or Monte-Carlo) framework. Fogaras et al. [8] propose to index the results of random walks, for faster query processing. Nevertheless, it incurs excessive space consumption for large graphs. Later, Lofgren et al. propose FastPPR [16]. Similar to BiPPR [15] and HubPPR [30] mentioned in Section 2.2, the key of FastPPR is to employ a bidirectional estimator that combines the sampling from the source s and reverse frontier discovery from the target t. However, FastPPR is less efficient compared with BiPPR, as indicated in [15], [14].

All-Pair PPR and Others. There are also studies on all-pair PPRs [28], [26], [29]. However, their complexities for the all-pair PPR queries are all $\Omega(n^2)$, making them prohibitive to be applied to very large graphs. Besides, many problems that are related to PPR queries have significantly different problem settings. Therefore, there are different technical challenges compared with the batch one-hop PPR queries addressed in this paper. For example, Refs. [6], [10], [25], [17] study the problem of answering PPR queries in a distributed cluster of machines. Refs. [20], [32] considers efficient tracking of the PPRs in evolving graphs. Ref [5] discusses generalizations of Personalized PageRank with node-dependent restart. Ref [4] also proposes a PageRank estimation using the entire random walk path. This version of random walks gives the same computation complexity as the ending node based estimation.

3 Our Solution

In this section, we present efficient algorithms for the batch one-hop PPR queries. We introduce a tightened lower bound for one-hop PPR values in Section 3.1, which is fundamental to our proposed Baton algorithm described in Sections 3.2∼3.4. In Section 3.5, we introduce a parallel algorithm for the batch one-hop PPR queries, which improves practical efficiency by multiple CPU cores.

3.1 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 our estimation of $\pi(s, v)$, 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 (All the proofs of lemmas can be found in Section 6).

Lemma 1. For any node s and any out-neighbor t of s, we have $\pi(s, t) \geq \alpha(1 - \alpha)/d(s)$, where $d(s)$ denotes the out-degree of s.

The above lower bound, albeit simple, could be exploited to significantly reduce the overhead of batch one-hop PPR queries. For example, consider the FORA algorithm (discussed in Section 2.2), which answers any single-source PPR query from a node s in $O\left(\sqrt{\frac{m\log(1/p_f)}{\epsilon^2\delta}}\right)$ 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 2, 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 HubPPR, so as to reduce their expected time complexities for batch one-hop queries. Particularly, we denote the residue threshold for conducting reverse push by r_{max}, which is similar in spirit to the threshold for the forward push. For each pair (s, t), the original HubPPR should sample $O\left(\frac{3r_{max}\log(2/p_f)}{\epsilon^2\delta}\right)$ random walks from the source node s, where δ is set to $O(\frac{1}{n})$ typically [15], [30]. Similar to the aforementioned improvement for FORA, we can replace δ with $\alpha(1 - \alpha)/d(s)$ so that there requires only $O\left(\frac{3d(s)r_{max}\log(2/p_f)}{\epsilon^2\alpha(1-\alpha)}\right)$ random walks started at the source node s to ensure ϵ relative error in the estimation of $\pi(s, v)$, as long as v is an out-neighbor of s. Hence the expected cost of random walks is $O\left(\frac{3d(s)r_{max}\log(2/p_f)}{\epsilon^2\alpha(1-\alpha)}\right)$. Also, by the techniques provided in [15], the reverse push has an expected cost $O(\bar{d}/r_{max})$. Therefore, by Lagrange optimization techniques to optimize the total cost $O\left(\frac{3d(s)r_{max}\log(2/p_f)}{\epsilon^2\alpha(1-\alpha)}\right) + O(\bar{d}/r_{max})$, one can set $r_{max} = O\left(\epsilon\sqrt{\frac{m\alpha(1-\alpha)}{d(s)n\log(1/p_f)}}\right)$ to achieve the minimum, giving the expected cost of computing a single-pair $\pi(s, t)$ to be $O\left(\sqrt{\frac{md(s)\log(1/p_f)}{n\epsilon^2\alpha(1-\alpha)}}\right)$. The total expected cost of batch one-hop PPR is therefore $O\left(\sum_{s\in S}d(s)\sqrt{\frac{md(s)\log(1/p_f)}{n\epsilon^2\alpha(1-\alpha)}}\right)$.

As shown in our experiments (in Section 4), the above respective improved FORA and HubPPR perform significantly better than their original versions in processing batch

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                     $\hat{\pi}(s,t) = \hat{\pi}(s,t) + \frac{1}{K(s)}$
```

one-hop PPR. Nevertheless, their time complexities are still unsatisfactory due to the $\sqrt{m}$ factor (for the improved FORA) and $\sum_{s\in S} d(s)^{1.5}$ factor (for the improved HubPPR). In the following, we will address this issue with a new solution whose time complexity is independent of m and only linear to $\sum_{s\in S} d(s)$, while it has a much better performance than the improved FORA and HubPPR.

3.2 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

$$r(s,u) > \frac{d(u)}{\alpha \cdot K(s)}, \tag{2}$$

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

$$r(s,u) > d(u) \cdot r_{max}, \tag{3}$$

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 could lead to significantly improved performance.

First, it is known that when the "push condition" in Equation 3 is adopted, the forward push method (i.e., Algorithm 1) runs in $O(1/r_{max})$ time [2]. FORA relies on this result to bound the computation cost of its forward push phase [31], and hence, it also adopts Equation 3. As such, changing the push condition from Equation 3 to Equation 2 invalidates the time complexity analysis in [31], and requires new analytical results to be derived for the revised forward push method.

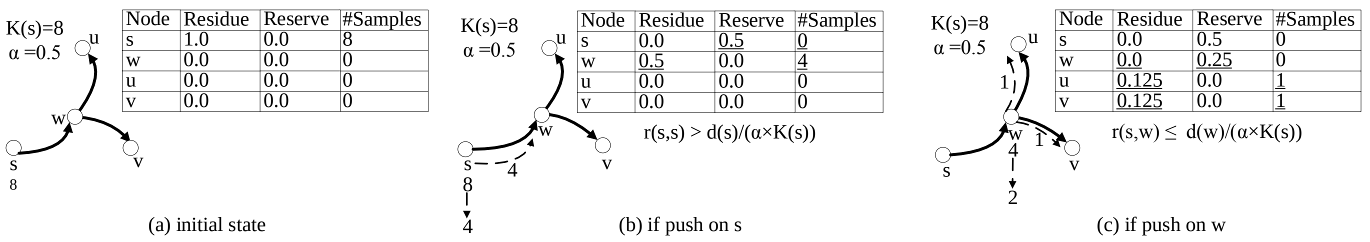
Second, the reason that Baton uses the push condition in Equation 2 is that it helps Baton achieve improved asymptotic performance by striking a better balance between forward push and random walks. To explain, let us consider a *generic* version of Baton (denoted that is identical to Algorithm 3, except that the push condition in Line 3 is changed arbitrarily). In other words, the algorithm performs a number of push steps based on certain push condition, and then generates random walks following Lines 7-12 in Algorithm 3 to estimate one-hop PPR. (Note that both FORA and Baton are special cases of this generic approach.) We first establish the accuracy guarantee of this algorithm (referred to as *Generic-Baton*).

Lemma 2. For all $s \in S$ and all out-neighbors v of s, Generic-Baton returns an estimated PPR $\hat{\pi}(s,v)$ that satisfies Equation 1 with at least $1 - p_f$ probability.

By Lemma 2, all instantiations of Generic-Baton provide the accuracy guarantee that we require for batch one-hop PPR queries. As such, a natural question is: which instantiation could offer us a high efficiency? To answer this question, we need to examine the cost and benefit of each push step, since the push condition is the only differentiating factor in different Generic-Baton instantiations.

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. We illustrate this strategy with an example.

Example 1. Figure 1 illustrates several typical steps of Baton. Suppose that $K(s) = 8$ and $\alpha = 0.5$. If no push step is conducted, then Baton requires $K(s) = 8$ random walks from the source node s, as illustrated in Figure 1a. Now consider that we are to decide whether or not to perform a push step on s, i.e., the only node with non-zero residue. In that case, we compute the value of $\frac{d(s)}{\alpha \cdot K(s)}$, and compare it against $r(s,s)$. Observe that $r(s,s) = 1$, while $\frac{d(s)}{\alpha \cdot K(s)} = \frac{1}{0.5 \times 8} = 0.25$, and hence,



Node	Residue	Reserve	#Samples
s	1.0	0.0	8
w	0.0	0.0	0
u	0.0	0.0	0
v	0.0	0.0	0

(a) initial state

Node	Residue	Reserve	#Samples
s	0.0	0.5	0
w	0.5	0.0	4
u	0.0	0.0	0
v	0.0	0.0	0

(b) if push on s

Node	Residue	Reserve	#Samples
s	0.0	0.5	0
w	0.0	0.25	0
u	0.125	0.0	1
v	0.125	0.0	1

(c) if push on w

Fig. 1: Illustration of Baton's push strategy.

$r(s,s) > \frac{d(s)}{\alpha \cdot K(s)}$, due to which the push step will be performed, and the result is illustrated in Figure 1b. In particular, an α fraction of s's residue is added to its reserve, resulting in an increased reserve (from 0 to $\alpha(=0.5)$). After that, the residue of node s is reset to 0, meaning that at this state, no random walk from s is needed. As a tradeoff, we need $8 \cdot (1-\alpha) = 4$ more random walks from w, which is the only out-neighbor of s. Next, we consider whether a push step is needed on w. From Figure 1b, we have $\frac{d(w)}{\alpha \cdot K(s)} = \frac{2}{0.5 \times 8} = 0.5 \geq r(s,w)$. In that case, Baton would not apply a push step on w. □

Based on the push strategy of Baton, we establish its time complexity as follows.

Lemma 3. Given a set S of source nodes, Baton runs in $O\left(\sum_{s \in S} \frac{d(s)\log(1/p_f)}{\epsilon^2}\right)$ expected time.

Note that the expected complexity of Baton is near-optimal. To explain, observe that a batch one-hop PPR query should return an estimation of $\pi(s,v)$ for each $s \in S$ and each out-neighbor v of s. As there exist $O\left(\sum_{s \in S} d(s)\right)$ such node pairs (s,v), the time complexity of any batch one-hop PPR algorithm is $\Omega(\sum_{s \in S} d(s))$. In comparison, Baton's expected time complexity is only a factor of $O(\frac{\log(1/p_f)}{\epsilon^2})$ larger, which is logarithmic to n for a typical setting of $p_f = 1/n$ and $\epsilon = 0.5$ [15], [30], [31].

Apart from the above result concerning Baton's efficiency for worst-case inputs, we can also show that for any input graph G, the computation cost of Baton is a close to a *local minimum* in a certain sense. In particular, let $\mathcal{P} = \{u_1, \ldots, u_h\}$ denote the sequence of nodes on which Baton performs push steps in its forward push phase, and $\mathcal{R}$ be the set of random walks conducted after that. In that case, we have $|\mathcal{R}| = r_{sum}(\mathcal{P}) \cdot K(s)$, where $r_{sum}(\mathcal{P})$ denotes the sum of all nodes' residues after the push steps in $\mathcal{P}$. We quantify the cost of the forward push phase as $cost(\mathcal{P}) = \sum_{u \in \mathcal{P}} d(u) \cdot c_p$, and the cost of the random walk phase as $cost(\mathcal{R}) = |\mathcal{R}| \cdot c_r$, where c_p and c_r are two constants that denote the cost of changing a node's residue or reserve and the expected cost of a random walk, respectively. In other words, the total cost of Baton is quantified as

$$cost(\mathcal{P},\mathcal{R}) = \underbrace{\sum_{u \in \mathcal{P}} d(u) \cdot c_p}_{\text{cost of } \mathcal{P}} + \underbrace{r_{sum}(\mathcal{P}) \cdot K(s) \cdot c_r}_{\text{cost of } \mathcal{R}} \quad (4)$$

We have the following result.

Lemma 4. Let $\mathcal{P}^*$ be the sequence of push steps performed by an instantiation of Generic-Baton given G, and $\mathcal{R}^*$ be the set of random walks that it generates. Further let $\mathcal{P}$ and $\mathcal{R}$ be those of Baton. If $\mathcal{P}$ is a prefix of $\mathcal{P}^*$ or $\mathcal{P}^*$ is a prefix of $\mathcal{P}$, then $cost(\mathcal{P}^*,\mathcal{R}^*) \geq \min\{1, \frac{c_r}{c_p}, \alpha c_p, c_r\} \cdot cost(\mathcal{P},\mathcal{R})$.

Intuitively, Lemma 4 shows that if an instantiation of Generic-Baton conducts strictly more (or strictly less) forward push than Baton, then its total cost cannot be lower than Baton's by more than a constant factor. This indicates that Baton's computation cost is close to a local minimum.

3.3 Reusing of Random Walks

Recall that for each node $s \in S$, Baton needs to generate a number of random walks from those nodes with non-zero residues at the end of the forward push phase. In addition, a node $u \in V$ may have non-zero residues in the forward push phases from two different nodes $s_1, s_2 \in S$. In that case, we can improve Baton's efficiency by *reusing* random walks from u when processing s_1 and s_2. For example, suppose that we first process s_1 and generate x random walks from u during the random walk phase, after which we proceed with s_2 and find that the random walk phase requires $y > x$ random walks from u. Then, instead of performing y new random walks from u, we take the x walks that were generated during the processing of s_1, and add only $y - x$ new random walks. This approach could considerably reduce the computation overhead of Baton due to the reduced cost of random walk generation. The only issue is that, for each node $u \in V$, we need to record all random walks that have generated from u (so as to facilitate reusing in subsequent steps), which leads to some space overhead. Fortunately, the total space cost incurred is only $O(m)$. To explain, recall that for any node $s \in S$, Baton's random walk phase generates $r(s,u) \cdot K(s)$ from any node u, where $r(s,u)$ denotes u's residue after the forward push phase ends. Meanwhile, Baton's push condition ensures that $r(s,u) \leq d(u)/(\alpha \cdot K(s))$. As a consequence, $r(s,u) \cdot K(s) \leq d(u)/\alpha$, i.e., we need to record at most $d(u)/\alpha$ random walks from u. Therefore, the total number of random walks required is $\sum_{u \in V} d(u)/\alpha = m/\alpha$, leading to a space overhead of $O(m)$.

3.4 Vertex Cover based Optimization

We present an optimization technique for online gaming user networks. These networks are typically undirected graphs. In addition, the source set S in a batch one-hop PPR may contain $O(n)$ nodes (which is the case for the online gaming platform discussed in Section 1). For such graphs, we can further improve the performance of Baton by exploiting the characteristics of undirected PPR. In particular,

Algorithm 4: Baton-VC(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 Compute a vertex cover $C$ of $G$
2 Invoke Baton($G$, $C$, $\epsilon$, $p_f$, $\alpha$)
3 for each $s \in \{S \setminus C\}$ do
4     for each neighbor $v$ of $s$ do
5         $\hat{\pi}(s,v) = \hat{\pi}(v,s) \cdot \frac{d(v)}{d(s)}$
```

our solution is based on the following result from existing work [3].

Lemma 5 ([3]). For any two nodes s and v in an undirected graph, $\pi(s,t) \cdot d(s) = \pi(t,s) \cdot d(t)$.

By Lemma 5, given an estimation $\hat{\pi}(t,s)$ of $\pi(t,s)$, we can easily obtain an estimation $\hat{\pi}(s,t)$ of $\pi(s,t)$ by setting

$$\hat{\pi}(s,t) = \hat{\pi}(t,s) \cdot \frac{d(t)}{d(s)}. \tag{5}$$

In addition, if $\hat{\pi}(t,s)$ ensures ϵ relative error, then $\hat{\pi}(s,t)$ also guarantees ϵ relative error. This motivates us to answer a batch one-hop PPR query with S containing $O(n)$ nodes as follows. First, we compute a vertex cover C of V using the standard 2-approximation algorithm [22]. Then, we perform a batch one-hop query using C as the source set, which provides us an approximate PPR $\hat{\pi}(c,v)$ for any $c \in C$ and any v that is c's neighbor. Note that these PPR approximations include the results for any one-hop PPR query with $s \in S \cap C$. Meanwhile, for any node $s \in S \setminus C$, its neighbors must be all in C (since C is a vertex cover). Accordingly, for any neighbor t of s, we obtain an approximate PPR $\hat{\pi}(s,t)$ using Equation 5. Algorithm 4 presents the pseudo-code of this approach.

However, the above approach has one potential issue when one takes the value of δ for further pruning. Recall that in the definition of the batch one-hop PPR query (Definition 1), only the PPRs larger than δ are considered to be guaranteed with an ϵ relative error. Hence, in the case where $\delta > \alpha(1-\alpha)/d(s)$, one can use δ directly as the lower bound of the one-hop PPR. When Baton incorporates this kind of δ-based pruning, Baton only ensures ϵ relative error for those PPR values no smaller than δ. In other words, if for nodes $c \in C$ and $v \in V$ we have $\pi(c,v) < \delta$, then $\hat{\pi}(c,v)$ do not guarantee ϵ relative error. In that case, if we use Equation 5 to compute $\hat{\pi}(v,c) = \hat{\pi}(c,v) \cdot \frac{d(c)}{d(v)}$, then $\hat{\pi}(v,c)$ would not ensure ϵ relative error, either. This would be an issue when $\pi(v,c) > \delta$, since we are supposed to make sure that all PPR at least δ should have at most ϵ relative error.

To remedy this, one can do the following. When answering one-hop PPR query for a node v which is not in the vertex cover C, we first check whether all the PPRs of its one-hop neighbors are of the accuracy indicated in Equation 1. In particular, if for any of v's neighbor c, we have $\alpha(1-\alpha)/d(c) \geq \delta$, then we are sure that any estimated $\hat{\pi}(c,v)$ is of the accuracy guarantee in Equation 1. Then, using the aforementioned approach to compute $\hat{\pi}(v,c)$ using $\hat{\pi}(c,v)$ maintains the same accuracy.

TABLE 2: 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
Tencent-x1	26.1M	485.6M	undirected	Tencent graphs
Tencent-x2	50.1M	792.0M	undirected	Tencent graphs
Tencent-x3	58.2M	1.1B	undirected	Tencent graphs
Tencent-x4	74.3M	1.5B	undirected	Tencent graphs

3.5 Multi-Core Parallelization

Parallelizing Baton with multiple cores is relatively simple since the one-hop queries for different source nodes in S are independent. In our implementation, we put the $|S|$ one-hop queries into a queue, and create one thread per core to process the queries one by one. For load balancing, we order the $|S|$ queries in descending order of the degrees of their source nodes, since the processing cost of a query is linear to the degree of the source node (see Lemma 3). In particular, we have the following results.

Lemma 6. Suppose that the last query processed by the i-th thread takes $t(i)$ seconds, and that the i-th thread accomplishes all of its queries in $T(i)$ seconds. Then,

$$\max_{1 \leq i \leq q} T(i) - \min_{1 \leq j \leq q} T(j) \leq \max_{1 \leq p \leq q} t(p)$$

where q is the total number of threads.

Intuitively, the lemma states that the maximum workload difference among any two threads is bounded by the maximum time required by the last queries assigned to the threads. This explains why we order the queries in descending order of their estimated costs.

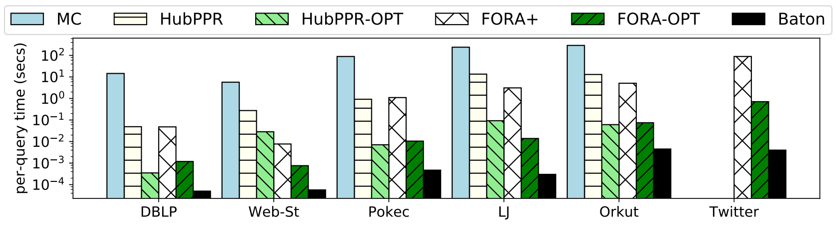
4 Experiments

In this section, we experimentally study the efficiency and accuracy of Baton, compared with the state-of-the-art methods, on various public benchmark datasets and Tencent datasets. We also evaluate the performance of parallel version of Baton introduced in Section 3.5. Finally, we present the results of an A/B test we conducted at Tencent platform, which demonstrate the effectiveness of the PPR based mechanism compared with other methods.

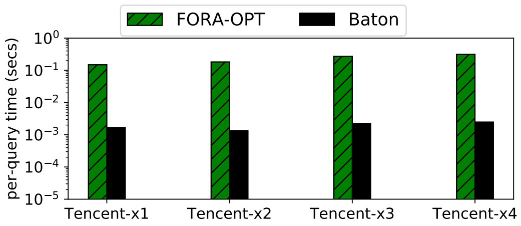
4.1 Settings

Datasets. We performed experiments on 10 real graphs. Among them, DBLP, Stanford Web (abbreviated as Web-St), Pokec, LiveJournal (abbreviated as LJ), Orkut, Twitter are used in recent works [15], [30], [31] as benchmark datasets for evaluating the efficiency and accuracy of PPRs. Tencent-x1 $\sim$ Tencent-x4 are four user networks from four representative Tencent games. The details of the datasets are shown in Table 2.

Methods and queries. We compare Baton with MC [8], HubPPR [30], FORA+ [31], and the respective optimized versions of the state-of-the-art algorithms FORA-OPT



(a) Benchmark public graphs.



(b) Tencent graphs.

Fig. 2: Efficiency.

and HubPPR-OPT using the tightened accuracy bounds (Lemma 1). FORA-OPT and HubPPR-OPT are the improved methods we described in Section 3.1. We name the method FORA-OPT (resp. HubPPR-OPT) to distinguish it with the original FORA (resp. HubPPR). We also show the improvement of multi-core Baton (introduced in Section 3.5) over Baton (using a single core). Note that, when we compare Baton with other baseline algorithms, we always report single-core performance for all the methods for fairness, as there are no existing parallelized versions of the baseline algorithms. The implementations of HubPPR [30] and FORA+ [31] are obtained from their respective authors. We implement the other algorithms by ourselves. 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 previous work [16], [15], [30], [31], we set $\delta = 1/n$, $p_f = 1/n$, and $\epsilon = 0.5$ by default. Our experiments are conducted on a Linux machine with an Intel 2.6GHz CPU and 64GB memory.

4.2 Performance

We first investigate the performance on public datasets. Figure 2 (a) shows the per-query efficiency of the methods. We do not report 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, on all the datasets. 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.

TABLE 3: 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

As we analyzed in Section 3.1, Baton employs an optimized mechanism to minimize the overall cost. To further illustrate this point by experiments, we summarize the average costs of forward push phase and random walk phase for FORA-OPT and Baton on the representative datasets in Table 3. 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 3.2, is reflected by the number of random walks. From Table 3 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 2 (a), 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 3.1.

To further investigate Baton's performance on Tencent graphs, we compare the performance between FORA-OPT and Baton. For clarify, we only remain the closest competitor FORA-OPT from now on, because 1) MC, HubPPR and FORA have been shown to be significantly slower than FORA-OPT (see Figure 2 (a)) and fail to efficiently handle batch one-hop PPR query on large graphs; 2) HubPPR-OPT fails to scale to billion-edge graphs while many Tencent graphs are of billion-edge scale.

As shown in Figure 2 (b), Baton also performs significantly better than the best competitor FORA-OPT on Tencent graphs, by around two orders of magnitude. This means the superiority of Baton is robust to various pub-

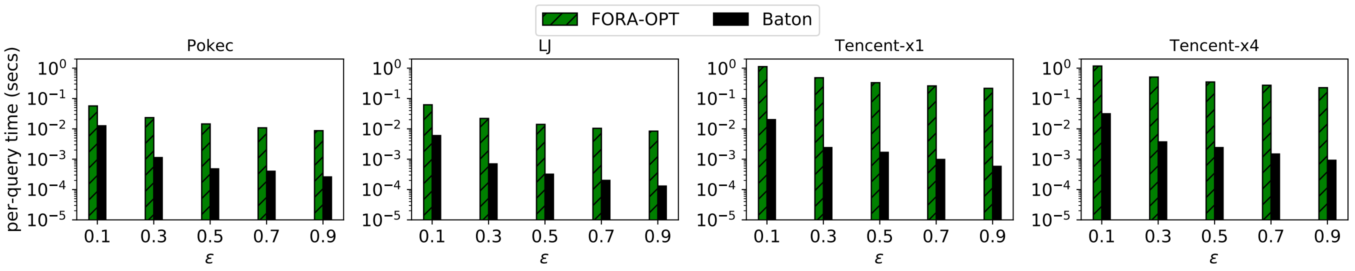


Fig. 3: Varying accuracy guarantee ϵ.

lic graphs and Tencent graphs, which exhibit significantly different characteristics (e.g., average degree, degree distributions).

Figure 3 shows that the performance of Baton is consistently better than FORA-OPT, with respect to various values of relative error ϵ, on four representative datasets. As expected, all the methods have better performance when ϵ is set to larger values, e.g., 0.9, since fewer pushes and random walks are required to achieve the required ϵ relative error. Fewer pushes and random walks, however, also typically makes it more difficult to balance the costs of forward push and random walks. Interestingly, our results show that the relative improvement of Baton over FORA-OPT is more significant when ϵ is large, demonstrating that in this case the push condition employed in Baton is much more effective than that of FORA-OPT. Take Tencent-x4's case as an example, Baton runs 37.8 times faster than FORA-OPT for $\epsilon = 0.1$, and 246.7 times faster when $\epsilon = 0.9$.

Accuracy. Recall that the main application of the batch one-hop PPR queries is about ordering the one-hop neighbors in terms of their PPR scores. A classic measure of the ordering quality is the Normalized Discounted Cumulative Gain (NDCG) [7], which is frequently employed to measure the difference between the returned ordering and the actual ordering (e.g., [23]). To compute the NDCG for a method $\mathcal{M}$, we sample 100 source nodes, and for each of them we compute the ordering of its one-hop neighbors based on the computed PPR scores using $\mathcal{M}$. With the NDCG formulae, this ordering is compared against the actual ordering based on the ground truth of PPR scores [2]. If the estimated ordering is close to the actual ordering, then the NDCG accuracy is close to 100% (We refer interested reader to [7], [23] for more details). Table 4 shows Baton's NDCG accuracies, comparing to FORA-OPT, the closest competitor in terms of efficiency and scalability. We perform the tests on 5 of our datasets (We omit the other datasets due to the prohibitive time required to compute the ground truth). The test shows that both Baton and FORA-OPT have very high NDCG accuracies.

Effect of Vertex Cover. We evaluate the effectiveness of vertex cover based approach, i.e., Algorithm 4. As mentioned in Section 3.4, the approach helps reduce the cost when $|S| = O(n)$. We therefore test the case where $S = V$, on two representative undirected graphs, DBLP and Orkut. We

2. Following [30], [31], the ground truth of PPR scores is computed by power iteration method. We set the absolute PPR error of power iteration method to be 10^{-9}.

TABLE 4: NDCG accuracy.

Dataset	DBLP	Web-St	Pokec	LJ	Orkut
FORA-OPT	100%	100%	99.8%	100%	100%
Baton	100%	100%	100%	100%	100%

TABLE 5: Speed-up ratio V.S. number of cores.

Datasets	5 cores	10 cores	15 cores
Twitter	4.8	9.0	12.1
Orkut	4.5	8.5	11.4

compute the total running time of Baton (Algorithm 3) and Baton-VC (Algorithm 4). By employing the techniques of vertex cover, the total running time for the batch one-hop queries for DBLP (resp. Orkut) has improved 1.52 (resp. 1.4) times respectively. The improvement is expected, as Baton-VC can reduce up to half of the cost by employing Lemma 5.

Mulit-Core Baton. We evaluate the performance of the multi-core Baton in Section 3.5. We run the algorithm for a batch one-hop query on two largest public graphs, i.e., Twitter and Orkut, with a randomly selected source set S such that $|S| = 10^6$. Table 5 shows the speed-up ratio of the performance of multi-core Baton compared with the single-core Baton. The results show that the speed-up ratio is in general reversely proportional to the number of CPU cores used, demonstrating that the workloads on each core are quite well balanced and the computation resources of each core are highly utilized.

4.3 A/B Test for Effectiveness of PPR

We have conducted an A/B test on the Tencent platform, which demonstrates that the PPR based method (mentioned in Introduction) is more effective than the other four methods we considered for attracting back the inactive game users. Each method $\mathcal{M}^*$ is tested as follows. First, for each inactive user s, the platform inspects her friends in the social network G formed by the game users, and identifies the ones who are active and important to s where the importance is measured by $\mathcal{M}^*$. 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. The following are the five methods we consider to measure the importance of a friend v (with respect to s).

TABLE 6: Effectiveness in A/B test.

Methods	Click rate C	Through rate T
PPR	**39.94%**	**3.91%**
TAP	37.88%	3.27%
PR	36.77%	3.16%
Degree	36.86%	3.10%
Proximity	36.93%	3.12%

- **PPR.** The importance of v equals to PPR(s, v) in G.
- **TAP (Topical Affinity Propagation) [27].** Measures the importance of v based on the influence of v to s using TAP. This method is the one originally employed by the Tencent platform before the PPR method is introduced.
- **PR (Global PageRank).** The importance of v is measured by the global PageRank score of v in G.
- **Degree.** The importance of v is based on the number of friends v has in G.
- **Proximity.** The importance of v is measured by the total number of the monthly gaming interactions between s and v, where the interactions include three activities: 1) game battle; 2) chatting and 3) reward sharing.

The A/B test is conducted on a game with about 220 million players, among which around 42 million monthly active players are identified to participate in the test to send invitations to their inactive friends. The test has a duration of 18 days, which are from April 4 to April 21 in 2018. Two classic metrics including the click rate C and through rate T are considered, where C is the percentage of the users who click the invitation link among those who view the invitation messages, and T is the percentage of the users who futher log in the game through the invitation link among those who click the invitation link.

The results are shown in Table 6. The PPR based method is more effective than all the others in both click rate and through rate. The improvement is significant for business. Consider that there are around 180 million inactive users in the game. Based on the result shown in Table 6, replacing the TAP based method with the PPR based method would attract back roughly $180M \cdot (39.94\% \cdot 3.91\% - 37.88\% \cdot 3.27\%) \approx 5.8 \times 10^5$ more inactive users.

One reason for the less effectiveness of PR and Degree is that they are more of global measures than measures that are personalized to s. Personalization, however, is crucial in our application. For example, a friend user v with a large degree in G can be popular among all users but this does not indicate that v is necessarily the most important to s. Therefore PR and Degree are not as effective as a personalized measure like PPR. PPR is also superior to the method Proximity as the latter suffers from the data sparsity. In the game user network we tested, each user s only has gaming interaction history with around 30% of her friends. As a result, for the remaining roughly 70% friends of s, the method Proximity cannot distinguish their importance with respect to s. This lack of information renders the Proximity method to be less satisfied. Further, we discover that the invitation senders that are identified for an inactive user s by PPR are quite different from those identified by TAP. Particularly, the senders suggested by PPR have about 15.3% more monthly gaming interactions with s on average, compared with the senders identified by TAP. This hints that the senders identified by the PPR are more close to s in gaming compared with those suggested by TAP, which could be a reason for the superiority of PPR over TAP.

5 Conclusion

In this paper, we conducted a comprehensive study on the batch one-hop PPR queries, which is a significant problem we encountered in deploying the applications on the Tencent online gaming platform with massive user bases. The motivation of the batch one-hop PPR is the observation that the PPR based mechanism has a significantly better effectiveness in attracting back the inactive game users, but its prohibitive computational cost hinders its application. Particularly, employing the existing state-of-the-art PPR algorithms to address batch one-hop PPR queries fail to scale to large graphs. To address this issue, 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 up to 3 orders of magnitude faster than the state-of-the-art algorithms. We also investigate the solutions for the parallelization of Baton. This significant efficiency improvement makes the deployment of the PPR based mechanism feasible.

6 Proofs

Proof of Lemma 1. Consider any random walk that starts from s. The walk has $(1-\alpha)/d(s)$ probability to move to t in the first step, after which it has α probability to stop. Therefore, the walk has at least $\alpha(1-\alpha)/d(s)$ probability to terminate at t, which proves the lemma.

Proof of Lemma 2. We consider the case for a source node s. Ref. [31] has shown the following result: If we are given a source node s, the reserves $\pi^{\circ}(s,\cdot)$ and residues $r(s,\cdot)$ resulted from any forward push process started at s, then we can achieve the PPR accuracy indicated in Equation 1 for *every node* $u \in V$ with $\pi(s,u) \geq \delta$, through the following procedures: 1) for each node $v \in V$, generate $r(s,v)K(s)$ random walks from v, where $K(s) = \frac{(\frac{2}{3}\epsilon+2)\log(2/p_f)}{\epsilon^2\delta}$. 2) After finishing all the random walks, for each node u, estimate $\hat{\pi}(s,u) = \pi^{\circ}(s,u) + c(u)/K(s)$, where $c(u)$ is the number of random walks that end at u.

With the above result, recall that *generic-baton* follows the above procedure but with a different value of $K(s)$ (see Line 2 in Algorithm 3). That is, *generic-baton* sets $K(s)$ to $\frac{(\frac{2}{3}\epsilon+2)d(s)\log(2/p_f)}{\epsilon^2\alpha(1-\alpha)}$ instead of $\frac{(\frac{2}{3}\epsilon+2)\log(2/p_f)}{\epsilon^2\delta}$. This modification of $K(s)$ is validated by Lemma 1. To explain, Lemma 1 says that the PPR of any one-hop neighbor of s is at least $\alpha(1-\alpha)/d(s)$. Therefore, in the above procedure, we can replace δ with $\alpha(1-\alpha)/d(s)$ without losing the accuracy guarantees for the one-hop PPRs of s. This results in $K(s) = \frac{(\frac{2}{3}\epsilon+2)d(s)\log(2/p_f)}{\epsilon^2\alpha(1-\alpha)}$, which is employed by *generic-baton*. Hence the lemma holds.

Proof of Lemma 3. For a source node s, the PPR computation initiated by Baton does not increase the complexity of directly conducting random walks from s. The reason is, when a push step is applied on node u, the number of random walks is reduced by $O(\alpha \cdot r(s,u) \cdot K(s))$, while incurring a cost $O(d(u))$. Since the expected complexity of a random walk is $O(1)$, the cost reduced by a push is therefore expected $O(\alpha \cdot r(s,u) \cdot K(s))$. Based on the push condition of Baton, we have $d(u) < \alpha \cdot r(s,u) \cdot K(s)$. Therefore, the complexity will not increase because of the pushes. On the other hand, if no pushes is conduct, the cost is $O(K(s)) = O(\frac{d(s)\log(1/p_f)}{\epsilon^2})$. Summing the cost over every node in S gives $O(\sum_{s\in S} K(s)) = O(\sum_{s\in S} \frac{d(s)\log(1/p_f)}{\epsilon^2})$.

Proof of Lemma 4.

Case 1. $\mathcal{P}^*$ is a prefix of $\mathcal{P}$. We first compare $cost(\mathcal{P})$ and $cost(\mathcal{P}^*)$. Let $\mathcal{P}^\circ = \mathcal{P} - \mathcal{P}^*$, then,

$$cost(\mathcal{P}) = cost(\mathcal{P}^*) + cost(\mathcal{P}^\circ) = cost(\mathcal{P}^*) + \sum_{u\in\mathcal{P}^\circ} d(u)\cdot c_p \tag{6}$$

Meanwhile, after the pushes in $\mathcal{P}^*$ have been applied, further pushes on node $u \in \mathcal{P}^\circ$ still satisfy $r(s,u) > \frac{d(u)}{\alpha\cdot K(s)}$ (Based on the condition of Baton), which gives

$$d(u) < \alpha \cdot r(s,u) \cdot K(s) \tag{7}$$

Next, we compare $cost(\mathcal{R}^*)$ and $cost(\mathcal{R})$. When a push step is applied on a node $u \in \mathcal{P}^\circ$, the sum of residues is reduced by $\alpha \cdot r(s,u)$. Hence,

$$cost(\mathcal{R}^*) = cost(\mathcal{R}) + \sum_{u\in\mathcal{P}^\circ} \alpha\cdot r(s,u)\cdot K(s)\cdot c_r \tag{8}$$

It then follows that,

$$\begin{aligned}
&\frac{cost(\mathcal{P}^*,\mathcal{R}^*)}{cost(\mathcal{P},\mathcal{R})} = \frac{cost(\mathcal{P}^*)+cost(\mathcal{R}^*)}{cost(\mathcal{P})+cost(\mathcal{R})}\\
&\text{(By } Eqn.\ 6)\\
&= \frac{cost(\mathcal{P}^*)+cost(\mathcal{R}^*)}{cost(\mathcal{P}^*)+\sum_{u\in\mathcal{P}^\circ}(d(u)\cdot c_p)+cost(\mathcal{R})}\\
&\text{(By } Eqn.\ 8)\\
&= \frac{cost(\mathcal{P}^*)+cost(\mathcal{R})+\sum_{u\in\mathcal{P}^\circ}(\alpha\cdot r(s,u)\cdot K(s)\cdot c_r)}{cost(\mathcal{P}^*)+\sum_{u\in\mathcal{P}^\circ}(d(u)\cdot c_p)+cost(\mathcal{R})}\\
&\text{(By } Eqn.\ 7)\\
&> \frac{cost(\mathcal{P}^*)+cost(\mathcal{R})+\sum_{u\in\mathcal{P}^\circ}(\alpha\cdot r(s,u)\cdot K(s)\cdot c_r)}{cost(\mathcal{P}^*)+cost(\mathcal{R})+\sum_{u\in\mathcal{P}^\circ}(\alpha\cdot r(s,u)\cdot K(s)\cdot c_p)}
\end{aligned} \tag{9}$$

Now, if $c_r \ge c_p$, then the R.H.S. of Equation 9 is at least 1; if $c_r < c_p$, then R.H.S. of Equation 9 equals

$$\begin{aligned}
&\frac{\frac{c_r}{c_p}\cdot\frac{c_p}{c_r}(cost(\mathcal{P}^*)+cost(\mathcal{R}))+\frac{c_r}{c_p}\sum_{u\in\mathcal{P}^\circ}(\alpha\cdot r(s,u)\cdot K(s)\cdot c_p)}{cost(\mathcal{P}^*)+cost(\mathcal{R})+\sum_{u\in\mathcal{P}^\circ}(\alpha\cdot r(s,u)\cdot K(s)\cdot c_p)}\\
&\frac{c_r}{c_p}\cdot\frac{\frac{c_p}{c_r}(cost(\mathcal{P}^*)+cost(\mathcal{R}))+\sum_{u\in\mathcal{P}^\circ}(\alpha\cdot r(s,u)\cdot K(s)\cdot c_p)}{cost(\mathcal{P}^*)+cost(\mathcal{R})+\sum_{u\in\mathcal{P}^\circ}(\alpha\cdot r(s,u)\cdot K(s)\cdot c_p)}\\
&> \frac{c_r}{c_p}\cdot 1 = \frac{c_r}{c_p}
\end{aligned}$$

Hence, for case 1, $cost(\mathcal{P}^*,\mathcal{R}^*) \ge \min\{1, \frac{c_r}{c_p}\}\cdot cost(\mathcal{P},\mathcal{R})$.

Case 2. $\mathcal{P}$ is a prefix of $\mathcal{P}^*$. Suppose that at time t_0 Baton terminates its push phase, whose cost is denoted by C_0. Further suppose that when the push steps according to $\mathcal{P}$ have been finished, node u's residue is $r(s,u)$. Based on the push-termination condition (Line 3 in Algorithm 3), at time t_0 every node u would satisfy

$$d(u) \ge \alpha\cdot r(s,u)\cdot K(s) \tag{10}$$

where $K(s) = \frac{(\frac{2}{3}\epsilon+2)d(s)\log(2/p_f)}{\epsilon^2\alpha(1-\alpha)}$.

On the other hand, at time t_0 the cost of random walks attached in node u is $r(s,u)\cdot K(s)$, which is the number of random walks to be performed from u (referred to Line 8 of Algorithm 3). Note that $\{\mathcal{P}^*,\mathcal{R}^*\}$ is a push-extended version of $\{\mathcal{P},\mathcal{R}\}$, such that it conducts any possible additional pushes after time t_0. Let us compare the costs of $\{\mathcal{P}^*,\mathcal{R}^*\}$ and $\{\mathcal{P},\mathcal{R}\}$. First,

$$cost(\mathcal{P},\mathcal{R}) = C_0 + \underbrace{\sum_{u\in V} r(s,u)\cdot K(s)\cdot c_r}_{\text{cost of random walks}} \tag{11}$$

As for $\{\mathcal{P}^*,\mathcal{R}^*\}$, we denote V^* as the set of nodes for which further push steps are performed after time t_0. Then, for the nodes in $V\backslash V^*$, no further push steps are performed on them. At the time when the push steps of $\mathcal{P}^*$ have been finished, the cost of random walks started at nodes in $V\backslash V^*$ is at least $\sum_{u\in\{V\backslash V^*\}} r(s,u)\cdot K(s)\cdot c_r$, because the residues of nodes in $V\backslash V^*$ will not decrease after time t_0. For node $u \in V^*$, $\{\mathcal{P}^*,\mathcal{R}^*\}$ incurs an additional push cost at least $d(u)\cdot c_p$, because one single push step performed on u costs $d(u)\cdot c_p$. Hence, the push cost due to the nodes in V^* is at least $\sum_{u\in V^*} d(u)\cdot c_p$. In a nutshell,

$$\begin{aligned}
cost(\mathcal{P}^*,\mathcal{R}^*) \ge\ & C_0 + \underbrace{\sum_{u\in V^*} d(u)\cdot c_p}_{\text{lower bound of cost}(\mathcal{P}^*)} + \underbrace{\sum_{u\in\{V\backslash V^*\}} r(s,u)K(s)c_r}_{\text{lower bound of cost}(\mathcal{R}^*)}\\
&\text{(by } Eqn.\ 10)\\
\ge & C_0 + \sum_{u\in V^*} \alpha r(s,u)K(s)c_p + \sum_{u\in\{V\backslash V^*\}} r(s,u)K(s)c_r\\
\ge & C_0 + \sum_{u\in V} \min\{\alpha c_p, c_r\} r(s,u)K(s)
\end{aligned} \tag{12}$$

By Equations 11 and 12,

$$\begin{aligned}
\frac{cost(\mathcal{P}^*,\mathcal{R}^*)}{cost(\mathcal{P},\mathcal{R})} &\ge \frac{C_0 + (\min\{\alpha c_p, c_r\})\sum_{u\in V} r(s,u)K(s)}{C_0+\sum_{u\in V} r(s,u)K(s)}\\
&\ge \min\{1, \alpha c_p, c_r\}
\end{aligned}$$

Hence, for case 2 we have

$$cost(\mathcal{P}^*,\mathcal{R}^*) \ge \min\{1,\alpha c_p, c_r\}\cdot cost(\mathcal{P},\mathcal{R})$$

Combining case 1 and case 2 gives

$$cost(\mathcal{P}^*,\mathcal{R}^*) \ge \min\{1,\frac{c_r}{c_p},\alpha c_p, c_r\}\cdot cost(\mathcal{P},\mathcal{R})$$

Proof of Lemma 6. Suppose that $T(i)$ is the largest among $\{T(1),\ldots,T(q)\}$, and $T(j)$ is the smallest among $\{T(1),\ldots,T(q)\}$. Then we need to prove $T(i)-T(j) \le t(k)$, where $t(k)$ is the maximum time required by the queries that are the last processed queries at each of the threads. In other words, $t(k) = \max_{1\le p\le q} t(p)$. We assume by contradiction that $T(i)-T(j) > t(k)$. Let Q_i be the last processed query at the i-th thread. Given that query Q_i runs in $t(i)$ seconds, we have

$$t(k) \ge t(i) \Rightarrow T(i)-T(j) > t(i) \Rightarrow T(j) < T(i)-t(i) \tag{13}$$

Equation 13 indicates that, when the j-th thread becomes idle at time $T(j)$, the i-th thread has not obtained query Q_i. As a result, there should be at least one queries (e.g.,

Q_i) that remain in the queue at time $T(j)$. It follows that at time $T(j)$ the j-th thread should gain another query from the queue for processing (the presumption here is that any reasonable parallelization mechanism won't let a thread be idle while there are still tasks to process). Now the discussion is divided into two cases: (1) if the j-th thread successfully obtains one query from the queue at time $T(j)$, then it contradicts to the fact that the j-th thread has accomplished all its queries at time $T(j)$. (2) if the j-th thread cannot obtain any query from the queue, then there must be other idle threads at time $T(j)$, and these idle threads obtain all the remaining queries in the queue, including Q_i, at time $T(j)$. Consequently, Q_i cannot be obtained by the i-th thread because at time $T(j)$ the i-th thread is still not idle. This contradicts to the fact that Q_i is the last processed query (i.e., the ending task) of the i-th thread. Therefore, in either case, violation ensues.

References

[1] R. Andersen, C. Borgs, J. Chayes, J. Hopcraft, V. Mirrokni, and S.-H. Teng, "Local computation of pagerank contributions," *WAW*, pp. 150–165, 2007.

[2] R. Andersen, F. Chung, and K. Lang, "Local graph partitioning using pagerank vectors," in *FOCS*, 2006, pp. 475–486.

[3] K. Avrachenkov, P. Gonçalves, and M. Sokol, "On the choice of kernel and labelled data in semi-supervised learning methods," in *WAW*, 2013, pp. 56–67.

[4] K. Avrachenkov, N. Litvak, D. Nemirovsky, and N. Osipova, "Monte carlo methods in pagerank computation: When one iteration is sufficient," *SIAM Journal on Numerical Analysis*, vol. 45, no. 2, pp. 890–904, 2007.

[5] K. Avrachenkov, R. Van Der Hofstad, and M. Sokol, "Personalized pagerank with node-dependent restart," in *WAW*, 2014, pp. 23–33.

[6] B. Bahmani, K. Chakrabarti, and D. Xin, "Fast personalized pagerank on mapreduce," in *SIGMOD*, 2011, pp. 973–984.

[7] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender, "Learning to rank using gradient descent," in *ICML*, 2005, pp. 89–96.

[8] 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.

[9] Y. Fujiwara, M. Nakatsuji, T. Yamamuro, H. Shiokawa, and M. Onizuka, "Efficient personalized pagerank with accuracy assurance," in *KDD*, 2012, pp. 15–23.

[10] T. Guo, X. Cao, G. Cong, J. Lu, and X. Lin, "Distributed algorithms on exact personalized pagerank," in *SIGMOD*, 2017, pp. 479–494.

[11] 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.

[12] H.-N. Kim and A. El Saddik, "Personalized pagerank vectors for tag recommendations: inside folkrank," in *RecSys*, 2011, pp. 45–52.

[13] 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.

[14] P. Lofgren, "Efficient algorithms for personalized pagerank," *arXiv preprint arXiv:1512.04633*, 2015.

[15] P. Lofgren, S. Banerjee, and A. Goel, "Personalized pagerank estimation and search: A bidirectional approach," in *WSDM*, 2016, pp. 163–172.

[16] P. A. Lofgren, S. Banerjee, A. Goel, and C. Seshadhri, "Fast-ppr: Scaling personalized pagerank estimation for large graphs," in *KDD*, 2014, pp. 1436–1445.

[17] S. Luo, "Distributed pagerank computation: An improved theoretical study," in *AAAI*, 2019.

[18] S. Luo, X. Xiao, W. Lin, and B. Kao, "Efficient batch one-hop personalized pageranks," *ICDE*, 2019.

[19] 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.

[20] N. Ohsaka, T. Maehara, and K.-i. Kawarabayashi, "Efficient pagerank tracking in evolving networks," in *KDD*, 2015, pp. 875–884.

[21] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[22] C. H. Papadimitriou and K. Steiglitz, *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

[23] Z. Qin, Y. Yang, T. Yu, I. Khalil, X. Xiao, and K. Ren, "Heavy hitter estimation over set-valued data with local differential privacy," in *CCS*, 2016, pp. 192–203.

[24] C. Ren, L. Mo, C. Kao, C. Cheng, and D. Cheung, "CLUDE: An efficient algorithm for LU decomposition over a sequence of evolving graphs," in *EDBT*, 2014.

[25] A. D. Sarma, A. R. Molla, G. Pandurangan, and E. Upfal, "Fast distributed pagerank computation," *Theoretical Computer Science*, vol. 561, pp. 113–121, 2015.

[26] K. Shin, J. Jung, S. Lee, and U. Kang, "Bear: Block elimination approach for random walk with restart on large graphs," in *SIGMOD*, 2015, pp. 1571–1585.

[27] J. Tang, J. Sun, C. Wang, and Z. Yang, "Social influence analysis in large-scale networks," in *KDD*, 2009, pp. 807–816.

[28] H. Tong, C. Faloutsos, and Y. Koren, "Fast direction-aware proximity for graph mining," in *KDD*, 2007, pp. 747–756.

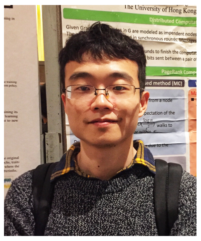
[29] H. Tong, C. Faloutsos, and J.-Y. Pan, "Fast random walk with restart and its applications," 2006.

[30] 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.

[31] 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.

[32] H. Zhang, P. Lofgren, and A. Goel, "Approximate personalized pagerank on dynamic graphs," in *KDD*, 2016, pp. 1315–1324.

[33] F. Zhu, Y. Fang, K. C.-C. Chang, and J. Ying, "Incremental and accuracy-aware personalized pagerank through scheduled approximation," *VLDB*, vol. 6, no. 6, pp. 481–492, 2013.



Siqiang Luo received the BSc and MS degrees from Fudan University in 2010 and 2013 respectively. He is a Ph.D. candidate in the Department of Computer Science, University of Hong Kong (HKU) under the supervision of Prof. Ben Kao and Dr. Reynold Cheng. His research interests are in the areas of adaptive optimization techniques, spatio-temporal data management, graph algorithms and parallel computing.

Xiaokui Xiao received the PhD degree in computer science from the Chinese University of Hong Kong in 2008. He is currently an Associate Professor at the National University of Singapore (NUS), Singapore. From 2009 to 2017, he was a faculty member at the Nanyang Technological University (NTU), Singapore. His research interests include data privacy, spatial databases, graph databases, and parallel computing. He is an associate editor of TKDE and VLDBJ.

Wenqing Lin Wenqing Lin received the PhD degree in Computer Science from Nanyang Technological University in 2015. Currently he is a Senior Researcher at the Game Data Mining Center of Tencent, China. His research interests include data mining, machine learning, and parallel computing.

Ben Kao received the BSc degree in computer science from the University of Hong Kong, in 1989, and the PhD degree in computer science from Princeton University, in 1995. He is currently a professor in the Department of Computer Science with the University of Hong Kong. From 1992 to 1995, he was a research fellow with Stanford University. His research interests include database management systems, data mining, real-time systems, and information retrieval systems.