Personalized PageRanks over Dynamic Graphs – The Case for Optimizing Quality of Service

Zulun Zhu
Nanyang Technological University
ZULUN001@e.ntu.edu.sg

Siqiang Luo
Nanyang Technological University
siqiang.luo@ntu.edu.sg

Wenqing Lin
Tencent
edwlin@tencent.com

Sibo Wang
The Chinese University of Hong Kong
swang@se.cuhk.edu.hk

Dingheng Mo
Nanyang Technological University
dingheng001@e.ntu.edu.sg

Chunbo Li
Nanyang Technological University
chunbo001@e.ntu.edu.sg

***Abstract*—We study the problem of Quality-of-Service (QoS)-Aware Personalized PageRank (PPR) computation. Existing studies mostly focus on improving the PPR query processing time. However, the query processing time alone may not reflect the service quality in real-world PPR-based systems. The query response time can be a more service-relevant measure in many applications such as the online game service of Tencent and the related-pin recommendation module of Pinterest. We make the first attempt at studying QoS-Aware PPR computation and present *Quota*, a system that adapts the state-of-the-art PPR algorithms to a given environment for minimizing query response time. Equipped with mathematical tools including queuing theory, algorithmic complexity analysis, and constrained optimization, *Quota* is designed to adapt itself to a wide spectrum of workloads. We conduct extensive experiments on real datasets and show that *Quota* can reduce the query response time compared with state-of-the-art PPR algorithms, often by a significant margin.**

I. Introduction

Given a source node s and a target node t in a graph, the Personalized PageRank (PPR) $\pi(s,t)$ reflects the probability that a random walk starting from node s terminates at node t. PPR is a fundamental proximity measure between two nodes in graphs, and it has been widely adopted in various applications such as the Whom-to-Follow service of Twitter [1], the game service of Tencent [2], and the related-pin recommendation of Pinterest [3]. As real-world graphs (e.g., in the aforementioned applications) are dynamically evolving with edge inserts or deletes, it becomes increasingly important to study efficient PPR queries on dynamic graphs [4].

When performing PPR queries in dynamically evolving graphs, it is an intrinsic requirement to simultaneously process PPR queries and data updates. The data update is not only about updating the graph but may also involve updating the index built to facilitate queries. For example, Pinterest uses PPR queries on its underlying user-item preference graph to support related-pin recommendations for users, and each page visit will trigger a PPR query. According to the reports [5], Pinterest has thousands of page visits per second, demonstrating a high query workload. Meanwhile, the extensive user base in Pinterest can lead to frequent updates in the preference graph, which happens in the same duration as queries, and ultimately forms a mixture workload of queries and updates. As another example, the online gaming service of Tencent [6] uses PPR values to measure the proximity between two game players, where the PPR query is computed on a game player network formed by players (nodes) and player interactions (edges). To avoid customer attrition, an incentive strategy based on PPR has been proven effective in Tencent [6]. Frequently, a PPR query is issued at the node of an active player (who regularly uses the service) to sort their proximity to other inactive players. Those highly-approximate but inactive players will receive an invite-back message from the active player. Meanwhile, while applying the incentive strategy, the player network evolves fast and leads to a mix-load of PPR queries and data updates. In summary, in real-world applications based on PPR measures, the workload often consists of a series of PPR queries and data updates, which may arrive in a stochastic manner.

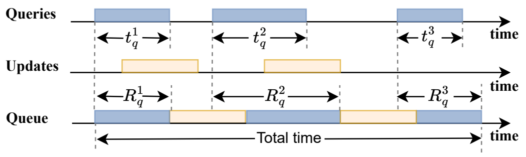


Fig. 1. Timeline of query and update arrivals, as well as the status of the queue.

In these applications, when the queries and updates arrive at the system faster than their processing speed, they line up and form a queue, as shown in Figure 1. The function of the queue is to ensure that the queries and updates are processed in a first-come-first-served (FCFS) manner, which is crucial to guarantee the query accuracy [4], scheduling fairness [7], and user fairness [8], [9]. In this queuing context, one important measure to be optimized is the *query response time*, which refers to the amount of time taken between the query's arrival at the system and the time at which the query's answer is computed, also illustrated as R_q in Figure 1. Query response time is not only determined by the exact query processing time t_q and the processing total time but is also influenced by the query/update queuing status. To explain, before a query can be executed, its previous updates on the graph or indexes have to be enforced, which can incur a significant cost. Hence, there exists the contention of CPU time between query and update processing, which often leads to an imbalance of computing resource allocation between queries and updates. As the contention intensifies, the user waiting time for queries can be extended, resulting in lengthy query response time for the users. In a high-load situation, the queue builds up quickly, and the response time of a query can be significantly exaggerated, impacting user satisfaction. Therefore, the query response time closely reflects the Quality-of-Service (QoS) to the user in a PPR-based application.

The problem: QoS-Aware PPR Computations. Despite the importance of queue-based PPR query response time optimization in practical applications, most of the existing PPR algorithms [10], [11], [12], [13], [14], [15], [16] focus on a stand-alone PPR query or update, as reviewed in Section III. However, the query response time does not solely depend on query or update processing time, but is more related to the behavior combination of PPR queries and data updates in a waiting queue, as we discussed earlier. In this paper, we make the first attempt at PPR computation with a focus on optimizing the query response time for single-source PPR (SSPPR) queries or top-k PPR queries over dynamic graphs, and we name the problem as QoS-Aware PPR Computation. In particular, given any PPR query and update arrival rates, we aim to design an algorithmic framework that can adapt itself to suit the workload in producing a reasonable average query response time.

QoS-Aware PPR computation is a non-trivial problem. First, directly applying the state-of-the-art PPR query algorithms (e.g., *Agenda* [4], *FORA* [13]) does not ensure a reasonable query response time, because they mainly optimize for a shorter query time at the expense of update time (e.g., updating index structures). As we have argued, faster query processing is not necessarily translated into a shorter query response time. Second, the workload of PPR queries and graph updates may change over time in recommendation systems. However, the mainstream PPR algorithms often entail many hyperparameters, making it difficult to set justifiable parameters under different workloads.

These challenges motivate us to design *Quota*, [1] a system that aims to optimize PPR query response time for a wide spectrum of query and update workload configurations. We emphasize several main design philosophies of *Quota*. First, our purpose is not to design a faster PPR query algorithm than existing ones which are already close to the optimum in many aspects due to the promising development in recent decades. Instead, given the maturity of the PPR algorithms, it can be more important to design an algorithmic framework to accommodate some state-of-the-art PPR algorithms as base algorithms and allow them to be transparently adapted for optimizing query response time. In this paper, we discuss three base algorithms: *Agenda* [4], *FORA* [13], and *SpeedPPR* [16], because *Agenda* [4] is the state-of-the-art dynamic PPR algorithm, *FORA* [13] is a representative PPR algorithm that motivates many later algorithms, and *SpeedPPR* is one of the most query-efficient PPR algorithms in static graphs. Second, the design of *Quota* integrates several different branches of mathematical tools including queuing theory, constrained optimization, and complexity analysis. Particularly, *Quota* abstracts the tuning parameters of a typical PPR query framework and builds a mathematical relationship between the parameters and the real cost. It then incorporates an in-depth queuing model to facilitate the transformation of the problem into a constrained optimization problem. We also design algorithms to solve the optimization problem to precisely determine the most suitable parameters for the given workload.

We summarize our main contributions as follows:

- **Problem Formulation.** We define the problem of QoS-Aware PPR optimization, which reflects the issues when applying PPR measures in practical recommendation systems. The problem is significantly different from the existing PPR optimization problems which optimize a stand-alone PPR query or update.
- **Auto-Configuration System.** We present *Quota*, a configuration system that can be deployed with some state-of-the-art PPR algorithms to optimize the query response time. *Quota* is an auto-configuration system to enhance the state-of-the-art approach in adaptability to various query and update workloads. *Quota* can be applied to four recent state-of-the-art algorithms, namely, *Agenda* [4], *FORA* [13], *SpeedPPR* [16], and *TopPPR* [17]. To the best of our knowledge, *Quota* is the first configuration system that aims to optimize the PPR query response time for PPR-based systems.
- **Reordering Algorithm.** In order to further optimize the query response time, *Quota* further allows violation of the FCFS queuing policy with controllable shuffling of the queries and updates, without affecting the accuracy guarantees. This forms the reordering algorithm named *Seed*.
- **Extensive Experiments.** We have conducted extensive experiments in various real datasets and settings. Our results demonstrate that *Quota* can achieve up to 86%, 40%, 34%, 50% and 33% shorter response time than the state-of-the-art PPR algorithms *Agenda*, *FORA*, *SpeedPPR*, *FORA-TopK* and TopPPR respectively, if deployed with the corresponding algorithm.

II. Problem Definition

A. PPR Queries

Let $G = (V, E)$ be a directed graph. Given a source node $s \in V$ and a probability $\alpha \in (0, 1)$, a random walk from s is a traversal in G where at each walk step it will move to a neighbor chosen uniformly at random with probability $1 - \alpha$, and otherwise terminate at the current node. The Personalized PageRank (PPR) value from s to t, denoted by $\pi(G, s, t)$, equals the probability that a random walk starting from s terminates at t. An important task of PPR computation is the single-source PPR query, defined as follows.

Definition 1. *(SSPPR queries) Given a source node s, a threshold $\delta \in (0, 1)$, an error bound ϵ, and a failure probability p_f, a Single-Source PPR (SSPPR) query returns an estimated PPR value $\hat{\pi}(G, s, t)$ for each vertex $t \in V$, such that for any $\pi(G, s, t) > \delta$, we have:*

[1] ***Qu**ality of Service **O**p**t**imization for Person**a**lized PageRank over Evolving Graphs

$$|\pi(G,s,t)-\hat{\pi}(G,s,t)| \leq \epsilon \cdot \pi(G,s,t) \quad (1)$$

holds with at least $1-p_f$ *probability.*

SSPPR is a popular form of query in dynamic graphs [11], [18], [4], and has been seen in burgeoning applications including query-tracking [19], [20], anomaly tracking [21], graph neural network [22], and parallel PPR processing [23] in evolving graphs. We consider SSPPR by default but our method can be extended to top-k PPR queries (see Section VIII-G).

B. Graph Updates

The real-world graphs are dynamic with continuous inserts and deletes of edges and vertices [24]. We note that the node inserts and deletes can be replaced by inserting and deleting the corresponding incident edges, and hence in this paper we focus on edge updates. We consider the edge arrival model in [11], [10]. Denote the initial graph as $G_0=(V_0,E_0)$, and edge updates $S_u=\{e_1,e_2,...,e_k,...\}$ arrive stochastically. Here, $e_i=(u,v)$ is the i-th edge update that will transfer the current graph G_{i-1} into G_i. If e_i already exists in G_{i-1}, e_i will be regarded as the delete from (u,v); otherwise e_i is an insert. Note that the insert of a new node u is linked with an update $e_i=(u,v)$ or $e_i=(v,u)$ for a certain node v. The delete of node u will be triggered immediately if u has no incident edges. We let $n_i=|V_i|$ and $m_i=|E_i|$. If the context is clear, we also use notations n and m for the numbers of nodes and edges in the *current graph*.

C. Query Response Time Optimization

Many online services (e.g., Pinterest, LinkedIn, and Tencent) incorporate the PPR measure in their user-recommendation module. Their back-end systems receive the PPR queries (abbr. queries) from the user side, and the underlying graph is subject to continuous updates (e.g., edge inserts or deletes). In other words, the back-end system is continuously receiving interleaving queries and updates, and each query or update has a timestamp. In a high-load situation, the system can build a waiting queue containing the arrived but unprocessed queries or updates, ordering by their arrival timestamps. To ensure query accuracy, a query result should be based on the graph that has executed the updates, which arrived before the query. In this paper, we put much higher priority on query response time for its direct reflection on the quality of service, from the perspective of user experience.

QoS-Aware PPR Optimization. Denote that the *response time* of a query is the amount of time taken between its arrival and finish, including the queuing time. Given a query arrival rate λ_q and update arrival rate λ_u (i.e., there are expected λ_q queries and λ_u updates arriving per second), we aim to minimize the average response time R_q of SSPPR queries where the query accuracy satisfies Eq. 1, i.e., ϵ relative error.

III. BACKGROUND

A. PPR Queries in Static Graphs

Push. *Forward Push* [25], [26] is a classic method for computing PPR. It is a local power iteration algorithm which passes the probability along the neighbors. Formally, the *Forward Push* conducted in Graph G maintains an estimated vector $\pi^\circ(G,s,t)$ of $\pi(G,s,t)$, given the source node s and any node $t\in V$. This estimated vector is also named *reserve value*. At the same time, it also maintains a residue value $r(G,s,t)$ for each $t\in V$. In each iteration, it chooses an arbitrary vertex t that satisfies $r(G,s,t)/d^{out}(G,t) > r_{max}$, where $d^{out}(G,t)$ is the out-degree of t. Then it performs a *Forward Push* operation on the current node t, where α fraction of residue $r(G,s,t)$ will be added to the reserve $\pi^\circ(G,s,t)$. The extra $(1-\alpha)$ fraction of residue $r(G,s,t)$ will be allocated uniformly to the neighbors of t. Hence, each neighbor of t will receive the residue value of $(1-\alpha)r(G,s,t)/d^{out}(G,t)$. Value $\pi^\circ(G,s,t)$ is the final estimate of $\pi(G,s,t)$. The detailed steps of *Forward Push* can be found in our technique report [27]. *Forward Push* entails a time complexity $O(\frac{1}{\alpha r_{max}})$ [26]. Similarly, *Reverse Push* [28] inherits the spirit of *Forward Push* and calculate PPR $\pi(G,s,t)$ from other nodes s to a target node t, which also entails a time complexity $O(\frac{1}{\alpha r^b_{max}})$ given the threshold r^b_{max}.

Push+Walk. To improve the efficiency, *Push+Walk* is proposed in *FORA* [13] and *ResAcc* [29], which run *Forward Push* starting from source node s, and then perform $K\cdot r(G,s,t)$ for each $t\in V$, where K is a parameter related to the estimation accuracy. The threshold r_{max} controls the degree of doing *Forward Push* against conducting random walks. The index-free *FORA* computes random walks online, whereas the index-based version *FORA+* precomputes all the random walks needed for higher efficiency. Another representative work that uses the same framework is *SpeedPPR* [16], which combines *Power Iteration* [30] and *Forward Push*. *SpeedPPR* achieves more promising results than *FORA* in some aspects including query time, accuracy as well as index size.

B. PPR Queries in Dynamic Graphs

In the dynamic graph scenarios, the PPR algorithms should provide the service for queries as well as updates. The *Query+Update* module raises new challenges for those methods which were mainly designed for improving the query efficiency. For example, many methods achieve high query efficiency by constructing an index. However, updating the graph requires updating the indexes as well, incurring a high update cost. To handle graph updates, existing literature can either **(a)** rebuild the pre-computed index when there comes an update; or **(b)** conduct a local update process to maintain the query accuracy [4], [10]. For example, *ApPPR* [11] proposes to conduct a local *reverse push* and *forward push* when answering an added edge, which will update the stored vectors for PPR computing. However, the focus of *ApPPR* is to track a fixed-source PPR query during graph updates and is not able to benefit our scenario which requires computing random PPR queries. Along the research direction, *Agenda* [4] is a recent state-of-the-art approach that performs the update only when needed, rendering it more robust when handling different query and update workloads.

Agenda [4]. Following the *Push+Walk* framework, *Agenda* takes the graph edge inserts and deletes into consideration and is the first work to consider the query and update sequence in real dynamic PPR scenarios. Different from the focus of this paper, *Agenda* aims to optimize the total cost of handling queries and updates. In particular, *Agenda* develops a graph update mechanism and applies the *Push+Walk*-based query in evolving graphs. Note that when there occur edge inserts or deletes, directly utilizing the obsolete index will cause unbounded errors. Hence, it may be required to update the index before a query to maintain the query accuracy. The novelty of *Agenda* is that, instead of regenerating the index immediately, *Agenda* updates the index *only when necessary*. Meanwhile, the update process also monitors a value for each node to quantify the maximum query error bound caused by using a stale index. With the aid of such new knowledge, *Agenda* can perform the query operations as well as the efficient updates.

While *Agenda* optimizes the total running time of queries and updates, our experiments (Section VIII) show that it is sub-optimal to optimize query response time. The main issue is that *Agenda* adopts a relatively rigid parameter setting. Hence *Agenda* determines the portions of forward push, random walks, and reverse push in a relatively inflexible manner, leading to sub-optimal performance in some situations.

C. The Research Gap

In summary, there still exists a gap between the current PPR algorithms (e.g., *FORA*[13], *Agenda* [4]) and optimizing the query response time in practical scenarios. First, the existing PPR algorithms target the overall running time or space overhead as the main goal, which neglects the practical queuing status consisting of queries and updates. Second, to achieve the optimal queuing status and query response time, the existing PPR algorithms can only empirically tune the hyperparameters to transform the query and update time. This empirical fine-tuning is inefficient and can easily lead to a sub-optimal solution. Furthermore, once the query or update arrival rate changes, this kind of artificial fine-tuning needs time-consuming calculation from scratch because the stale solution does not apply to the new workload, which further degrades the system efficiency. Based on these fundamental challenges, *Quota* is designed to coordinate the queuing status adaptively according to different query and update arrival rates, and further reduce the query response time.

IV. *Quota*

The goal of *Quota* is to determine the optimal hyperparameters that minimize query response time given query and update arrival rates. When tuning the hyperparameters of a PPR algorithm, different combinations of query time $\tilde{t}_q$ and update time $\tilde{t}_u$ are often observed, resulting in various query response times for different query and update arrival rates. For instance, recent state-of-the-art methods such as *Agenda* [4], *FORA* [13], and *SpeedPPR* [16] have query and update costs that are inversely related. Specifically, both *SpeedPPR* and *FORA* have an index-based and an index-free version, with the former having a low query cost and a high update cost and the opposite for the latter. Interestingly, the degree of leaning towards which version can be tuned by hyperparameters, which sheds light on the optimization potential in different contexts. Similarly, *Agenda* also has hyperparameters that trade query cost for update cost or vice versa. Hence, a natural question arises: how to tune the hyperparameters to optimize query response time under diverse workloads and algorithmic contexts? This motivates the design of *Quota* to tackle the QoS-Aware PPR problem.

To fully unleash the potential of hyperparameter tuning, *Quota* adopts a drastically different design philosophy compared with the prior art. In the existing works, the hyperparameter settings are *workload unconscious*, meaning that a single set of hyperparameters is used for all possible workloads. However, this lack of flexibility can lead to suboptimal performance when faced with challenging workloads. Further, the hyperparameters are often determined by minimizing the query complexity. For example, *FORA* [13] sets the hyperparameter r_{max} to be $1/\sqrt{\alpha m K}$ to optimize its query complexity $O\left(1/(\alpha r_{\max}) + m r_{max} K\right)$. However, such an optimization neglects the hidden constants in the complexity, which potentially impacts the actual optimum. When the hidden constants come into play, purely optimizing the complexity expression falls short in searching for the optimal parameters.

To circumvent these issues, *Quota* takes a novel approach by modeling the QoS-Aware problem using queuing theory and constrained optimization techniques that take into account both workloads and hidden constants. That is, by modeling response time using query/update time functions $\tilde{t}_q(\cdot)/\tilde{t}_u(\cdot)$ that incorporate hyperparameters and hidden constants, as well as workload characteristics such as query and update arrival rates, we can search for the optimal hyperparameter setting that balances update and query performance, ultimately leading to better query response times. This idea motivates the design of *Quota*, which involves the following three steps (also shown in Figure 2):

•**Step 1-Modeling the expected query/update cost.** Given a PPR algorithm (with query and update functions) and query/update arrival rates, we first model the query and update cost functions based on the hyperparameters that shall be tuned, via a complexity analysis.

•**Step 2-Modeling response time.** We use queuing theory to model the query response time with respect to the mean query and update costs computed in Step 1, which can be further expressed by a set of hyperparameters.

•**Step 3-Optimizing the expected query response time.** Then, when considering the essential constraints, we minimize the query response time with a set of constraints, which is efficiently solved by constrained optimization techniques.

In the following sections, we will focus on Steps 2 and 3, which are the core components of *Quota*. We will address Step 1, which involves the refinement of different PPR algorithms, in Section V. Figure 2 provides a helpful overview of the *Quota* framework, outlining the roadmap for each section.

A. Modeling Response Time

One important property of a queuing system is the **stability**. A system is stable when the input workload is less than the

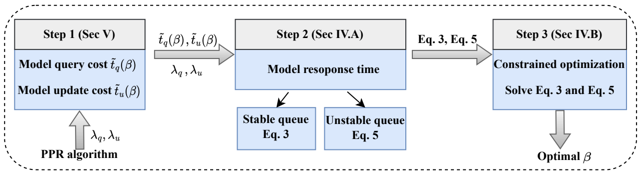


Fig. 2. Linking each section to the *Quota* framework.

system's service capacity. Let the query arrival rate and update arrival rate be λ_q and λ_u, respectively. The stability of a queue in dynamic PPR is formally defined as:

Definition 2. *The queue is stable only when $\lambda_q \tilde{t}_q + \lambda_u \tilde{t}_u < 1$, where $\tilde{t}_q$ and $\tilde{t}_u$ are the average query time and update time.*

A stable queue secures a bounded expected response time within an extended period of time. On the contrary, if the input workload equals or exceeds the system's capacity (e.g., $\lambda_q \tilde{t}_q + \lambda_u \tilde{t}_u \geq 1$), it forms an unstable queue and the response time will be continuously increasing. Due to the disparity between stable status and unstable status, our discussion is divided into the corresponding two parts.

1) Stable Status: We aim to express the average response time of a stable query by the query and update arrival rates, the mean query and update processing time, and their variances. In particular, let $\boldsymbol{\beta} \in \Lambda$ be the chosen hyperparameter vector, where $\Lambda = \Lambda_1 \times \ldots \times \Lambda_w$ is the w-dimensional hyperparameter space. We assume that the mean query and update time $\tilde{t}_q$ and $\tilde{t}_u$ can be expressed as two functions of $\boldsymbol{\beta}$. Here we abuse the notations $\tilde{t}_q$ and $\tilde{t}_u$ so that they can also represent functions. By the results in [31], one can estimate the average query response time based on $\tilde{t}_q$, $\tilde{t}_u$, and their respective coefficients of variation[2] CV_q and CV_u. In particular,

$$R_q(\boldsymbol{\beta}) = \frac{\lambda_u \tilde{t}_u^2(\boldsymbol{\beta})(1 + CV_u^2) + \lambda_q \tilde{t}_q^2(\boldsymbol{\beta})(1 + CV_q^2)}{2\left(1 - \lambda_q \tilde{t}_q(\boldsymbol{\beta}) - \lambda_u \tilde{t}_u(\boldsymbol{\beta})\right)} + \tilde{t}_q(\boldsymbol{\beta}) \quad (2)$$

It is important to note that other estimates in [31] that are under different models are also applicable in our framework. Due to space constraints, we just discuss one kind of estimate here. Based on Eq. 2, it can be inferred that the expected response time of a query, denoted by $R_q(\boldsymbol{\beta})$, is prone to elevate with an increase in the system's workload, which is represented as $\lambda_q \tilde{t}_q(\boldsymbol{\beta}) + \lambda_u \tilde{t}_u(\boldsymbol{\beta})$. Intuitively, a high workload close to 1 may lead to congestion in the queue, thereby deteriorating the performance of the system.

In this paper, for simplicity we only focus on the tuning process of mean query and update time because tuning the coefficient of variation is usually insignificant compared with tuning mean query/update times.

Constraints. The optimization of the optimal response time is then translated into computing the minimum solution of the objective function $R_q(\boldsymbol{\beta})$. Compared with the ordinary extremum-seeking problem defined in the entire domain $\mathbb{R}^w$, the QoS-Aware PPR optimization should guarantee more constraint conditions. Particularly, we enforce two constraints for the hyperparameter vector $\boldsymbol{\beta}$:

[2]Coefficient of variation is a commonly used term to refer to the ratio between the standard deviation and mean.

- **Stability Constraint.** Given λ_q and λ_u, $\lambda_q \tilde{t}_q(\boldsymbol{\beta}) + \lambda_u \tilde{t}_u(\boldsymbol{\beta}) < 1$.
- **Search Space Constraint.** The solution of $\boldsymbol{\beta}$ exists in the search space Λ.

First, the necessity of maintaining system stability leads to the constraint $\lambda_q \tilde{t}_q + \lambda_u \tilde{t}_u < 1$, which gives $\lambda_q \tilde{t}_q(\boldsymbol{\beta}) + \lambda_u \tilde{t}_u(\boldsymbol{\beta}) < 1$ when the hyperparameter $\boldsymbol{\beta}$ is involved. Also, we need to guarantee that the hyperparameters are searched in the required space Λ determined by the PPR algorithms.

In a nutshell, *Quota* aims to search for the optimal hyperparameter solution $\boldsymbol{\beta}^*$ such that

$$\boldsymbol{\beta}^* = \arg\min_{\boldsymbol{\beta}} R_q(\boldsymbol{\beta}), \text{ subject to } C_i(\boldsymbol{\beta}) \leq 0, 1 \leq i \leq p, \quad (3)$$

where $C_i(\boldsymbol{\beta})$ is the constraint function and p is the number of constraint functions. Moreover, when $C_i(\boldsymbol{\beta}^*) \leq 0$, $\boldsymbol{\beta}^*$ satisfies stability constraint and search space constraint.

2) Unstable Status: Unfortunately, Luo et al. [31] do not give an estimate of the response time when the queue is unstable, where Eq. 2 becomes invalid to measure the response time. The main characteristic of unstable queue is that the high query or update arrival rates can break the stability constraint, which gives

$$\min_{\boldsymbol{\beta}}(\lambda_q \tilde{t}_q(\boldsymbol{\beta}) + \lambda_u \tilde{t}_u(\boldsymbol{\beta})) \geq 1. \quad (4)$$

In that case, the system is not stable even with the best parameter setting. Consequently, it leads to a crowded state such that Eq. 3 cannot serve as our objective function for reducing response time. While the response time in the unstable queue grows infinitely [32], we next prove that the response time of a *specific* query is still related to the mean query and update time in an expected manner, as shown in Lemma 1. All the proofs can be found in our technical report [27].

Lemma 1. *Suppose the queue is unstable given the query arrival rate λ_q and update arrival rate λ_u, i.e., $\lambda_q \tilde{t}_q + \lambda_u \tilde{t}_u \geq 1$. Let W_{N_q} denote the response time of N_q-th query, we have*

$$N_q^{-1} W_{N_q} = \frac{\lambda_q \tilde{t}_q + \lambda_u \tilde{t}_u - 1}{\lambda_q}, \textit{w.p.1 as } N_q \to \infty$$

Lemma 1 suggests that when N_q is sufficiently large, the response time of the N_q-th query is determined by the current index N_q and other queue indicators, such as λ_q, λ_u, $\tilde{t}_q$, and $\tilde{t}u$. Therefore, if Eq. 4 is satisfied, we can minimize $\rho(\boldsymbol{\beta}) = \lambda_q \tilde{t}_q(\boldsymbol{\beta}) + \lambda_u \tilde{t}_u(\boldsymbol{\beta})$ to reduce the response time of each query. This approach can be explained using queuing theory [32], [33], where $\rho(\boldsymbol{\beta})$ represents the traffic intensity or the system's level of congestion. By reducing $\rho(\boldsymbol{\beta})$, the system's workload is lessened, resulting in a lower response time.

In a nutshell, in the unstable status, *Quota* aims to search the optimal hyperparameter solution $\boldsymbol{\beta}^*$ such that:

$$\boldsymbol{\beta}^* = \arg\min_{\boldsymbol{\beta}} \rho(\boldsymbol{\beta}), \text{ subject to } C_i(\boldsymbol{\beta}) \leq 0, 1 \leq i \leq p. \quad (5)$$

Here we utilize $\rho(\boldsymbol{\beta})$ to replace the response time function $R_q(\boldsymbol{\beta})$ in the stable queues. It is evident that when using $\rho(\boldsymbol{\beta})$ as the response time measurement, the stability constraint cannot be met in this unstable queue. Thus for $C_i(\boldsymbol{\beta}^*) \leq 0$, $\boldsymbol{\beta}^*$ is only required to satisfy the search space constraint.

TABLE I
OVERVIEW OF THE COST FUNCTIONS, RESPONSE TIME FUNCTION, AND *Quota* OPTIMIZATION OBJECTIVE FUNCTIONS FOR EACH METHOD.

Method	**Query Cost** ($\tilde{t}_q$)	**Update Cost** ($\tilde{t}_u$)	**Objective Function (Sec. IV-A)**	
			Stable Status	Unstable Status
Agenda	$\frac{1}{r_{max}} \cdot \tau_1 + \frac{\lambda_u r_{max}(nr^b_{\max}+1)}{\lambda_q} \cdot \tau_2 + r_{max} \cdot \tau_3$	$\frac{1}{r^b_{max}} \cdot \tau_4 + \tau_5$	$\min R_q(r_{max}, r^b_{max})$ s.t. $0 < r_{max}, r^b_{max} < 1$, $\lambda_q \tilde{t}_q + \lambda_u \tilde{t}_u < 1$	$\min \rho(r_{max}, r^b_{max})$ s.t. $0 < r_{max}, r^b_{max} < 1$
TopPPR	$\frac{1}{r_{max}} \cdot \tau_1 + r_{max}(r^b_{\max})^2 \cdot \tau_2 + \frac{1}{r^b_{max}} \cdot \tau_3$	τ_4		
FORA	$\frac{1}{r_{\max}} \cdot \tau_1 + r_{max} \cdot \tau_2$	τ_3	$\min R_q(r_{max})$ s.t. $0 < r_{max} < 1$, $\lambda_q \tilde{t}_q + \lambda_u \tilde{t}_u < 1$	$\min \rho(r_{max})$ s.t. $0 < r_{max} < 1$
FORA+	$\frac{1}{r_{\max}} \cdot \tau_1 + r_{max} \cdot \tau_2$	$r_{max} \cdot \tau_3$		
FORA-TopK	$\frac{1}{r_{\max}} \cdot \tau_1 + r_{max} \cdot \tau_2$	τ_3		
SpeedPPR	$\log \frac{1}{r_{max} m} \cdot \tau_1 + r_{max} \cdot \tau_2$	τ_3		
SpeedPPR+	$\log \frac{1}{r_{max} m} \cdot \tau_1 + r_{max} \cdot \tau_2$	$r_{max} \cdot \tau_3$		

Algorithm 1: Online Constrained Optimization

```
Input    : The query and update arrival rates $\lambda_q$ and $\lambda_u$
Parameter: Formulation of $\tilde{t}_q(\boldsymbol{\beta})$ and $\tilde{t}_u(\boldsymbol{\beta})$; constraint
           functions $C_i(\boldsymbol{\beta})(1 \le i \le p)$
Output   : Optimal hyperparameter $\hat{\boldsymbol{\beta}}^*$
1  /* Objective function formulation */
2  Formulate the objective function $R_q(\boldsymbol{\beta})$ and $\rho(\boldsymbol{\beta})$ with
     $\tilde{t}_q(\boldsymbol{\beta})$, $\tilde{t}_u(\boldsymbol{\beta})$, $\lambda_q$ and $\lambda_u$;
3  /* Constrained optimization */
4  Initialize $\mu^0$ and $v_i^0$ $(1 \le i \le p)$; $k \leftarrow 0$;
5  if $\min(\lambda_q \tilde{t}_q(\boldsymbol{\beta}) + \lambda_u \tilde{t}_u(\boldsymbol{\beta})) \ge 1$ then
6      $\Phi^0(\boldsymbol{\beta}) = \rho(\boldsymbol{\beta}) + P^0(\boldsymbol{\beta})$;
7  else
8      $\Phi^0(\boldsymbol{\beta}) = R_q(\boldsymbol{\beta}) + P^0(\boldsymbol{\beta})$;
9  while not converge do
10     $\hat{\boldsymbol{\beta}}^* = \arg\min_{\boldsymbol{\beta}} \Phi^k(\boldsymbol{\beta})$; Update $v_i^{k+1} \leftarrow v_i^k + \mu^k C_i(\hat{\boldsymbol{\beta}}^*)$;
11     Update $\mu^k$ to $\mu^{k+1}$ [34]; $k \leftarrow k+1$;
```

B. Constrained Convex Optimization

In the optimization process for Eq. 3 or 5, we have two requirements. First, the constraints in Eq. 3 or 5 must be satisfied. We refer to these constraints as *Requirement (a)*. Second, the optimization process should be efficient, meaning that it should converge fast in a bounded number of iterations. We term this requirement as *Requirement (b)*. These two requirements can be met if we use the Augmented Lagrangian method [35], also see Algorithm 1. Let us first introduce the basic idea of the Augmented Lagrangian method, and then discuss why these two requirements are met.

The Augmented Lagrangian method is a penalty-based approach that replaces a constrained optimization problem with an unconstrained one. This is achieved by adding a series of penalty terms derived from the original constraints to the objective function. One of the key benefits of using the Augmented Lagrangian method is its guaranteed performance (refer to Theorem 2). The method employs an iteration-based approach to search for the optimal solution. In our case, let $P^k(\boldsymbol{\beta})$ denote the sum of penalty terms in the k-th iteration and $S(\boldsymbol{\beta})$ represent the objective function. Notably, the following unconstrained objective function is implemented in the k-th iteration:

$$\Phi^k(\boldsymbol{\beta}) = S(\boldsymbol{\beta}) + P^k(\boldsymbol{\beta}), \tag{6}$$

The penalty function $P^k(\boldsymbol{\beta})$ contains the penalty factor μ^k and Lagrange multiplier v_i^k in the k-th iteration. We follow the specific solution in [34] to initialize the penalty factor μ^0 as well as the Lagrange multiplier v_i^0 $(1 \le i \le p)$ (line 4 in Algorithm 1). Then we formulate the penalty function as:

$$P^k(\boldsymbol{\beta}) = \frac{\mu^k}{2} \sum_{i=1}^{p} \max(0, C_i(\boldsymbol{\beta}))^2 + \sum_{i=1}^{p} v_i^k \max(0, C_i(\boldsymbol{\beta}))$$

As for the objective function $S(\boldsymbol{\beta})$, we incorporate the response time function and traffic intensity function in Eq. 3 and 5 to address the stable and unstable status (line 5 to 8), which is formulated as:

$$S(\boldsymbol{\beta}) = \begin{cases} R_q(\boldsymbol{\beta}) & \text{if } \min(\lambda_q \tilde{t}_q(\boldsymbol{\beta}) + \lambda_u \tilde{t}_u(\boldsymbol{\beta})) < 1, \\ \rho(\boldsymbol{\beta}) & \text{otherwise.} \end{cases}$$

Note that searching the minimum of $(\lambda_q \tilde{t}_q(\boldsymbol{\beta}) + \lambda_u \tilde{t}_u(\boldsymbol{\beta}))$ can also be addressed by utilizing the Augmented Lagrangian method. Finally, we successfully transform the minimization problem of $S(\boldsymbol{\beta})$ with constraints into the unconstrained minimization problem of $\Phi^k(\boldsymbol{\beta})$, where the estimated solution $\hat{\boldsymbol{\beta}}^*$ in the k-th iteration (line 10) is searched by:

$$\hat{\boldsymbol{\beta}}^* = \arg\min_{\boldsymbol{\beta}} \Phi^k(\boldsymbol{\beta}).$$

In each iteration, the minimum of $\Phi^k(\boldsymbol{\beta})$ can be computed iteratively by the common L-BFGS-B optimizer [36], which offers the benefits of rapid convergence and minimal memory usage [37]. Then, we update μ^k to a larger value (lines 11) following the method in [34]. Moreover, in order to enhance the differentiable smoothness [38] of Eq. 6, Augmented Lagrangian will update v_i^k as:

$$v_i^{k+1} = v_i^k + \mu^k C_i(\hat{\boldsymbol{\beta}}^*),$$

where $\hat{\boldsymbol{\beta}}^*$ is the solution in current k-th iteration. Next we will demonstrate how we can meet our two requirements.

Meeting Requirement (a). To prove that the Augmented Lagrangian method satisfies the requirement (a), we slightly rephrase and cite a key theorem from the conclusion of [39]:

Theorem 1. *Assume that $S(\boldsymbol{\beta})$ and $C_i(\boldsymbol{\beta})$ are continuous functions, and that the constraint set $\{C_i(\boldsymbol{\beta}) \le 0\}$ is* *nonempty. Given μ^k is bounded, $0 < v_i^k < v_i^{k+1}$ for all k and $v_i^k \to \infty$, we search the solution $\hat{\boldsymbol{\beta}}^*$ for each iteration of Eq. 6. Then the limit point of sequence $\{\hat{\boldsymbol{\beta}}^*\}$ is a global minimum of the original problem in Eq.6.*

In summary, by combining the constraint functions, the optimal value of $\phi^k(\boldsymbol{\beta})$ will give precedence to the constraint conditions ($C_i(\boldsymbol{\beta}) \leq 0$). In other words, the final solution $\hat{\boldsymbol{\beta}}^*$ will converge to the circumstance where the constraints are not violated [40].

Meeting Requirement (b). The primary benefit of using the Augmented Lagrangian method is that as the penalty parameter μ^k increases, the quadratic term $\frac{\mu^k}{2}\sum_{i=1}^{p}\max(0, C_i(\boldsymbol{\beta}))^2$ in Eq. 6 will cause the function to become strongly convex, thereby ensuring a global minimum [41]. By the results of Lan and Monteiro [42]:

Theorem 2. *Given the error tolerance ϵ^*, it requires $O(1/\sqrt{\epsilon^*})$ iterations in the Augmented Lagrangian method to satisfy $|R_q^* - \hat{R}_q| < \epsilon^*$, where R_q^* is the true minimum value of response time and $\hat{R}_q$ is our estimated result.*

Theorem 2 states that the time complexity of optimization process is inversely related to the estimation accuracy, and is much smaller than that of the PPR query algorithms, as demonstrated in Section VIII-E. In summary, *Quota* can achieve the optimal solution of response time based on the tunable hyperparameters $\boldsymbol{\beta}$ with the global optimal value and a low time complexity.

V. Base PPR Algorithms

In this section, we analyze several state-of-the-art algorithms in literature and discuss how *Quota* can incorporate them. Recall the step 1 introduced in Section IV, we first model the expected query and update cost utilizing a refined complexity analysis. For ease of understanding, we provide the overview of cost function as well as the objective function of all listed methods in Table I. These methods include the state-of-the-art algorithm *Agenda* [4] targeting dynamic graph, as well as the static PPR algorithms index-free *FORA* [13], *SpeedPPR* [16], index-based *FORA+*, and *SpeedPPR+*. Additionally, we also incorporate two state-of-the-art top-k PPR methods *FORA-TopK* and TopPPR [17] in a similar fashion. Most of the results are directly extended from the literature, and we hence leave the detailed reference and analysis in our technique report [27].

In Table I, we rewrite the time complexity by incorporating the constant factor τ explicitly. Then we target two important tunable hyperparameters r_{max} and r_{max}^b ($0 < r_{max}, r_{max}^b < 1$), which are the thresholds in the *Forward Push* and *Reverse Push* phase (recall that in Section III). For example, the hyperparameter vector in a base algorithm *Agenda* is $\boldsymbol{\beta} = (r_{max}, r_{max}^b)$ and by the property of *Agenda*, tuning these two hyperparameters will not impact the worst-case accuracy guarantee with respect to the estimated PPR values.

Here we explain the rationale why *Quota* can effectively reduce response time of the listed PPR algorithms. Previous studies shown in Table I have predominantly focused on assessing the complexity of queries and updates, often overlooking the real-world queue status. This oversight can result in sub-optimal response time solutions. As a general framework, *Quota* can be easily applied to these representative algorithms and search the optimal solutions (e.g., in Eq. 3 and Eq. 5), which align with the queue status and the response time can be naturally reduced.

VI. Relaxing FCFS

A. Main ideas

In our previous discussions, we relied on the FCFS queuing principle as a fundamental requirement. However, there is potential to relax this constraint, as delaying certain minor updates may not significantly impact current queries [4]. This relaxation opens up opportunities for optimizing response times. Nonetheless, a challenge arises when we relax FCFS, as reordering queries and updates may raise concerns about maintaining bounded query accuracy. Notably, index-based algorithms (e.g., *Agenda*, *FORA+*, *SpeedPPR+* as analyzed in Table I) must consider index-updates, where the index refers to some precomputed random walks starting from each node. The delays of index-updates could lead to greater inaccuracy.

To tackle this challenge, we introduce *Seed*, [3] a queue-reordering algorithm that relaxes the FCFS queuing requirement while ensuring a bounded query error. The core idea of *Seed* involves elevating the processing priority of certain queries over some earlier-arrived updates. The algorithm continuously monitors the upper-bound of query error resulting from such adjustments to ensure an acceptable level of accuracy is maintained. By employing *Seed*, we strike a balance between optimizing response times and preserving query accuracy in the presence of reordered queries and updates.

B. Techniques

To formalize our reordering algorithm, we first show what will happen after reordering the queue elements. Let Q_i be a query on the graph G_i, and the k-th edge update after Q_i as U_k. Without loss of generality, we consider a case where there are k updates between Q_i and Q_{i+k} and the sequence in the queue can be formalized as $\Gamma = Q_iU_1U_2...U_kQ_{i+k}$. When processing with the original sequence, we obtain the estimated PPR $\hat{\pi}(G_{i+k}, s, t)$ upon the new graph G_{i+k} given the source node s and target node t. When we reorder the sequence as $\Gamma' = Q_iQ_{i+k}U_1U_2...U_k$, we can only process Q_{i+k} with the stale graph G_i to obtain the estimated PPR $\hat{\pi}(G_i, s, t)$. Then we express the overall inaccuracy of Q_{i+k} compared with the true PPR $\pi(G_{i+k}, s, t)$ as:

$$|\pi(G_{i+k}, s, t) - \hat{\pi}(G_i, s, t)| \leq \underbrace{|\pi(G_{i+k}, s, t) - \pi(G_i, s, t)|}_{\textit{Ordering Inaccuracy}} + |\pi(G_i, s, t) - \hat{\pi}(G_i, s, t)|.$$

In the above equation, $|\pi(G_i, s, t) - \hat{\pi}(G_i, s, t)|$ is decided by the base PPR algorithms. As a trade-off for query response time deduction, *Seed* will produce an additional *inaccuracy* caused by $|\pi(G_{i+k}, v, t) - \pi(G_i, v, t)|$ when we process Q_{i+k} before k edge updates $U_1U_2...U_k$. Considering all nodes we define the *ordering inaccuracy*.

Definition 3. (Ordering Inaccuracy). *Given the query source node s, graph G_i and the graph G_{i+k} after k edge updates, the Ordering Inaccuracy of s is:*

[3]***Se**quence R**e**or**d**ering*

$$\sigma(G_{i+k}, G_i, s) = \max_{t \in V_i} \left| \pi\left(G_{i+k}, s, t\right) - \pi\left(G_i, s, t\right) \right|.$$

To measure the maximum difference between PPR values given an edge update (u_{i+1}, v_{i+1}), we have the following lemma:

Lemma 2. *For an edge update (u_{i+1}, v_{i+1}), given any $s, t \in V_i$, the following holds:*

$$\sigma(G_{i+1}, G_i, s) \leq \frac{(e(G_i, s) - \alpha)(1 - \alpha(1 - \alpha))}{\alpha^2 d^{out}\left(G_{i+1}, u_{i+1}\right)},$$

where $e(G_i, u) = \frac{d^{out}(G_i,u) - \alpha(1-\alpha)\left(d^{out}(G_i,u)-1\right)}{d^{out}(G_i,u)}, u, v \in V_i$.

Similar to the derivation in Lemma 2, we can extend *Ordering Inaccuracy* $\sigma(G_{i+1}, G_i, s)$ to $\sigma(G_{i+k}, G_i, s)$ when incurring k updates:

$$\sigma(G_{i+k}, G_i, s) \leq \sum_{j=1}^{k} \frac{(e(G_{i+j-1}, s) - \alpha)(1 - \alpha(1 - \alpha))}{\alpha^2 d^{out}\left(G_{i+j}, u_{i+j}\right)}.$$

Upon integrating *Seed*, we present the pseudocode of *Quota* as Algorithm 2. To begin, Line 3 initializes a pending update queue denoted as U^p, initially empty. Next, *Seed* retrieves the next arrived operation, denoted as R, and examines its type. If R is an *Update*, the algorithm adds R to the pending queue (Line 6). If R is a *Query* with a specified source node s, *Seed* calculates the current error upper bound, denoted as $e_{sum}(s)$, accumulated by the pending queue (Line 10). If this upper bound exceeds a given threshold ϵ_r, *Seed* executes all updates stored in U^p and adds the new edges to the current graph (Line 12). Note that for index-based methods, the update includes updating their random walk indexes. By setting a reorder error threshold ϵ_r, *Seed* examines whether the query can be processed upon the stale graph, which puts much higher priority on query response time for its direct reflection on the quality of service. Naturally, Algorithm 2 returns the PPR with at most additional ϵ_r error. Moreover, we explore the overall performance and response time after applying *Seed*. Note that the overall performance can be effectively measured by the total processing time $\lambda_q \tilde{t_q} + \lambda_u \tilde{t_u}$. The following lemma shows that the *Seed* algorithm generally will not degrade the overall performance and the query response time, highlighting the benefits using *Seed*.

Lemma 3. *If the cost of each query and update remains unchanged after applying Seed, then the term $\lambda_q \tilde{t_q} + \lambda_u \tilde{t_u}$ also remains unchanged after applying Seed. Let $W^*_{N_q}$ and W_{N_q} be the response time of N_q-th query before and after applying Seed, we have $W_{N_q} \leq W^*_{N_q}$.*

Algorithm 2: Quota

```
Input  : Graph $G = (V_i, E_i)$, workload sequence $\Gamma$,
         reorder error threshold $\epsilon_r$, query and update
         arrival rates $\lambda_q$ and $\lambda_u$
Output: Result of workload sequence $\Gamma$
1  Invoke Algorithm 1 (Online Constrained Optimization)
     with $\lambda_q$ and $\lambda_u$;
2  /* Seed */
3  $U^p = \emptyset$;
4  while $R = next(\Gamma)$ is not empty do
5      if R is Update then
6          $U^p$.add($R$);
7      if R is Query then
8          Obtain the source node $s$ and set $e_{sum}(s) = 0$;
9          foreach Edge update $(u_{i+j}, v_{i+j})$ in $U^p$ do
10             $e_{sum}(s)$+=$\frac{(e(G_{i+j-1},s)-\alpha)(1-\alpha(1-\alpha))}{\alpha^2 d^{out}(G_{i+j},u_{i+j})}$;
11         if $e_{sum}(s) > \epsilon_r$ then
12             Execute the updates in $U^p$; $U^p = \emptyset$;
13         Execute the query $R$;
```

VII. Other Related Work

Hyperparameter search. Classical hyperparameter searching approaches such as Grid Search and Random Search exhaustively enumerate or randomly sample hyperparameter combinations to compute the optimal one. These methods, however, entail high computation cost especially when encountered a large search space [43]. Bayesian Optimization [44] collects the feedbacks from the possible hyperparameter combinations, and uses the Gaussian processes to build the objective function, which can compress the searching space and achieve higher efficiency than Random Search. However, like Grid Search and Random Search, Bayesian Optimization is not robust to changes in query and update arrival rates in dynamic PPR scenarios.

Queuing theory. The research on strategic decision-making in queuing systems can be traced back to the renowned Naor model [45]. Then the choice of individual arrival time to a congested bottleneck is extensively investigated in the research of [46]. This work delves into the "rush hour" problem [47], where customers must decide when to join the queue, while considering delay, earliness, and tardiness costs. Recent work [48] further explores the G/M/1 and M/M/1 models [49] where customers need to make a decision on whether to join or not while knowing the arrival time, and then provides the analysis of the optimal waiting time function.

PageRank algorithms. The PageRank algorithm, developed by Google [50], leverages the link structure of the web to assess the importance of web pages. The simplest linear algebraic method to acquire PageRank scores is the Power Iteration [51]. Several subsequent algorithms [52], [53], [54] combine various algebraic techniques, intelligently leveraging the characteristics of real-world graphs to enhance the efficiency of Power Iteration computation. Additionally, due to the properties of high efficiency and scalability, the Monte-Carlo-based methods [55], [56] are developed to conduct computation in large graphs.

More PPR algorithms. Except for the PPR algorithms mentioned in Section III, there are other studies on PPR computations that are based on different settings or approaches other than using Forward-Push-based or Random-Walk-based philosophy. These include the top-k PPR [17], [57], [58], [59], reverse-push-based method [60], [61], distributed-based computation [62], [63], [64], [65], parallel-processing [66], [23], [62], [59], query-tracking in dynamic graphs [19], [20], [12], [67], [68], graph-structure-based computation [69], [70], PPR with small decay factor [71], and one-hop PPR queries [2].

TABLE II
STATISTICS OF THE DATASETS. (K = 10^3, M = 10^6, B = 10^9)

Datasets	Nodes	Edges	Type	Source
Webs	281.9k	2.3M	directed	www.stanford.edu
DBLP	613.6k	2.0M	undirected	www.dblp.com
Pokec	1.6M	30.6M	directed	pokec.azet.sk
LJ	4.8M	69.0M	directed	www.livejournal.com
Orkut	3.1M	117.2M	undirected	www.orkut.com
Twitter	41.7M	1.5B	directed	twitter.com

As these algorithms mostly employ different settings or are dominated by the state-of-the-art PPR algorithms we consider, they are less relevant to our design of *Quota*, which aims to build a configuration system to optimize QoS for some recent state-of-the-art algorithms, particularly for those based on Forward-Push or Random-Walk procedures.

VIII. EXPERIMENTS

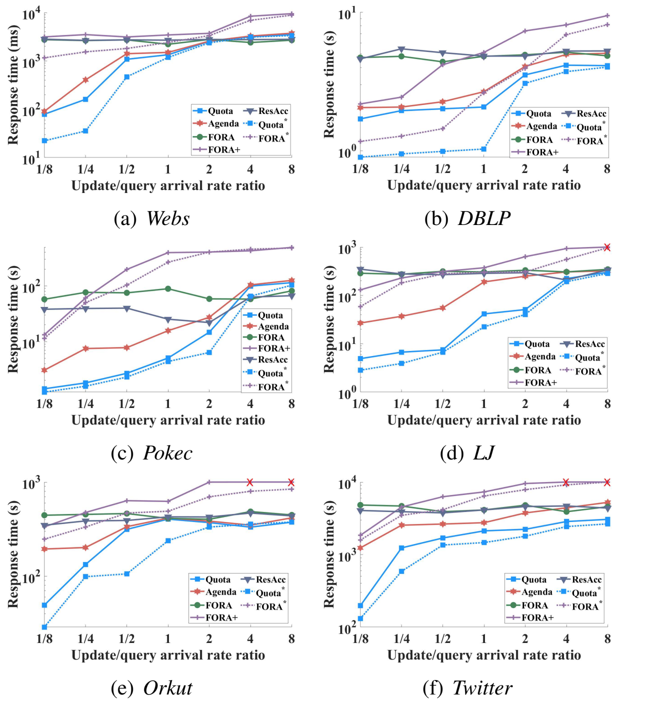
We comprehensively evaluate the QoS performance (i.e., query response time) of *Quota* when deployed on *Agenda*, referred to as *Quota-Agenda*. We also evaluate *Quota* deployed on *FORA*, *SpeedPPR* and *TopPPR*, to verify the generality of *Quota* designs. We conduct the experiments on a Linux machine with an Intel(R) Xeon(R) Gold 6326 CPU @ 2.90GHz and 256GB memory.

A. Baseline Methods

We conduct the experiments on *Quota-Agenda* and other four representative dynamic PPR algorithms, namely, *Agenda* [4], *FORA*, *FORA+* [13], and *ResAcc* [29]. *Agenda* [4] is the state of the art for dynamic PPR queries. *FORA+* and *ResAcc* are representative PPR algorithms for static graphs. We note that *Agenda* and *FORA+* are index-based, whereas the other two are index-free algorithms. We implement *Quota* on our own and adapt the baseline algorithms based on the public code provided by the authors. All the implementations are in C++ and follow the hyperparameter vectoring in previous work [57], [4], [13], where $\alpha = 0.2, \delta = \frac{1}{n}, p_f = \frac{1}{n}, \epsilon = 0.5$. For a fair comparison with other baselines that do not allow reordering, we first set reorder error threshold ϵ_r as 0 in our general performance comparison. We will perform the effectiveness analysis of ϵ_r in later Section VIII-H. Note that in the *Agenda* case, we target two hyperparameters r_{max} and r^b_{max}, aiming to search the optimal value starting from the default value $\bar{r}_{max} = \frac{1}{\alpha K}$, $\bar{r}^b_{max} = \frac{1}{n}$, and $K = \frac{(2\epsilon/3+2)\log(1/p_f)}{\epsilon^2\delta}$. We report all results using the averages of 10 runs.

B. Datasets and Queries

We evaluate the response time of dynamic PPR queries on six publicly available graph datasets: *Webs*, *DBLP*, *Pokec*, *LJ*, *Orkut*, and *Twitter*, as summarized in Table II. These datasets are chosen to represent different sizes and areas, such as citation networks, web graphs, and recommendation systems, to demonstrate the effectiveness of *Quota* in various graph application scenarios. In each scenario, the queries and updates are generated following the given query and update arrival rates. Specifically, we model the query and update arrivals as two Poisson processes within a time window. We randomly select the source node s among the query sequence from the current node set V_i for SSPPR queries. The number of edge updates follows a Poisson process with arrival rate λ_u, where each update (u, v) selects the two nodes u and v randomly from V_i.



(a) *Webs* (b) *DBLP*
(c) *Pokec* (d) *LJ*
(e) *Orkut* (f) *Twitter*

Fig. 3. Response time on 6 public datasets.

C. Arrival Rate and Constant Calculation

To simulate the dynamic workloads in real PPR scenarios, we take the query and update arrival rates as the system input, and measure the response time of different algorithms during a certain time window. We note that each PPR algorithm with which we compare *Quota* has various query and update processing time on different datasets. For example, the average query processing time of index-based *FORA+* on *Webs* and *Twitter* is 0.01 s and 103.10 s respectively. Hence, if the query arrival rate is larger than $\lambda_q = 100$ on *Webs* and $\lambda_q = 0.01$ on *Twitter*, the queue will be unstable. To consider both stable and unstable status for experimental purposes, we set the query and update arrival rates (λ_q, λ_u) and simulation time window T for all compared algorithms as follows.

For the small-scale graphs *Webs* and *DBLP*, we set $\lambda_q = 100$ and $\lambda_q = 10$ respectively and the simulation time window $T = 10s$. For larger datasets *Pokec*, *LJ*, and *Orkut*, we set $\lambda_q = 0.1$ and $T = 1000s$. Finally, for the largest graph *Twitter*, we set $\lambda_q = 0.01$ and $T = 10000s$. To check the robustness of *Quota* with different workloads, we set various update arrival rates to formulate the different characteristics of the query and update queue. In particular, we set $\lambda_u/\lambda_q = \{1/8, 1/4, 1/2, 1, 2, 4, 8\}$ for each dataset, which will cover a wide spectrum of interesting scenarios. For example, the experiments on *Webs* will be conducted with the query arrival rate $\lambda_q = 100$ and update arrival rate $\lambda_u = \{12.5, 25, 50, 100, 200, 400, 800\}$. Under such a specific setting, we formulate multiple competition statuses between the queries and updates in the queue. Finally, we remark that the values of τ are easy to be gauged as we can independently time the actual sub-process costs and infer the constants fairly precisely.

D. Response Time on Different Workloads

For all algorithms, we process dynamic workloads with different combinations of query and update arrival rates. We investigate the response time of *Quota-Agenda* (expressed as *Quota*) and other baselines on 6 public datasets, which are shown in Figure 3. Additionally, we integrate the *Seed* algorithm into *Quota* and *FORA+* when setting $\epsilon_r = 0.5$ (denoted as *Quota** and *FORA** respectively) and explore the performance after reordering. The red cross represents the cost that is above the maximum threshold. Below, we highlight our interesting observations.

(1) Effectiveness for reducing response time. As expected, *Quota-Agenda* outperforms baselines when processing various workloads on all the tested datasets. Compared with the original algorithm *Agenda*, *Quota-Agenda* entails a significantly shorter query response time when processing most workloads. We note that the optimization effect differs for different datasets. For example, when $\lambda_u/\lambda_q = 1/8$, *Quota* mildly reduces the response time of *Agenda*, by $(90.36 - 78.75)/90.36 = 12.85\%$ on *Webs* dataset, because the tested dataset is relatively small and the workload containing queries and updates is not that heavy. In such cases, the queue is short and the response time is small. In contrast, when the same ratio λ_u/λ_q is applied to the large *Twitter* dataset, the scale of the graph and the heavy workload will drastically reduce the query response time, which reaches $1236.45s$ for *Agenda*. With the support of *Quota*, *Quota-Agenda* fine-tunes the mean query and update time, achieving a much lower response time of $197.27s$. Additionally, we note that the response time for *Quota* and *FORA+* can be further reduced when we allow for additional errors and reorganize the sequences by invoking *Seed*. This is because queries receive higher priority when the impact of updates is not substantial, allowing an earlier query processing and thus a lower response time.

In summary, *Quota* effectively optimizes the queuing status to reduce the user's waiting time.

(2) Robustness to the evolving environments. We fix the query arrival rate λ_q for each dataset. The ratio λ_u/λ_q simulates the evolving workloads and working environments, which reflects different contention situations between queries and updates. The results in Figure 3 demonstrate that *Quota* automatically configures itself based on different workloads. We observe that *Quota-Agenda* outperforms the other algorithms in most cases. For example, in the largest dataset *Twitter*, *Quota-Agenda* achieves the response time deduction for at least 1200 seconds against *Agenda* when ratio λ_u/λ_q ranges from $1/8$ to 8. Interestingly, when the workload becomes extremely update-heavy (e.g., $\lambda_u/\lambda_q = 8$), *Quota-Agenda*'s performance converges to those of baseline algorithms such as *Agenda* and *FORA*. The main reason is that such an update-heavy situation is the most favorable case for these baseline algorithms that entail low update costs.

Except for the workloads shown in Figure 3, we further evaluate the performance of *Quota* in an evolving situation where the query and update rates fluctuate over time. Specifically, we track the response time on *DBLP* every 10s and apply the following 5 workload patterns: query-inclined ($\lambda_q = 10 \to 30, \lambda_u = 5$), balanced ($\lambda_q = 10 \to 15, \lambda_u = 10 \to 15$), update-inclined ($\lambda_q = 5, \lambda_u = 10 \to 30$), update-declined ($\lambda_q = 5, \lambda_u = 30 \to 10$), and query-declined ($\lambda_q = 30 \to 10, \lambda_u = 5$). As an illustration, in the query-inclined pattern, the intervals keeping stable rates follow a Poisson distribution with an average of 10s, while the query rate steadily rises from 10 to 30. We continuously monitor the rates and perform the hyperparameter search every 1s. As shown in Figure 4, we observe that *Quota* can adapt to this evolving situation and swiftly tune the hyperparameters, achieving reduced response times. Since we only tune the hyperparameters which have no impact on the worse-case accuracy guarantees, *Quota* consistently maintains a high level of accuracy in terms of PPR estimation results across all patterns. Besides, we also compare with *Quota-c*, a *Quota* version that neglects the hidden constants in the query and update cost functions. Our experimental result highlights that neglecting the influence of hidden constants result in sub-optimal outcomes in the hyperparameter search, leading to a significant increase in response time.

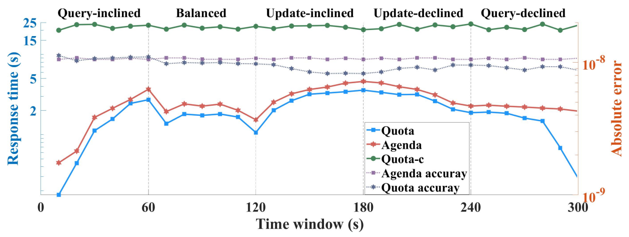
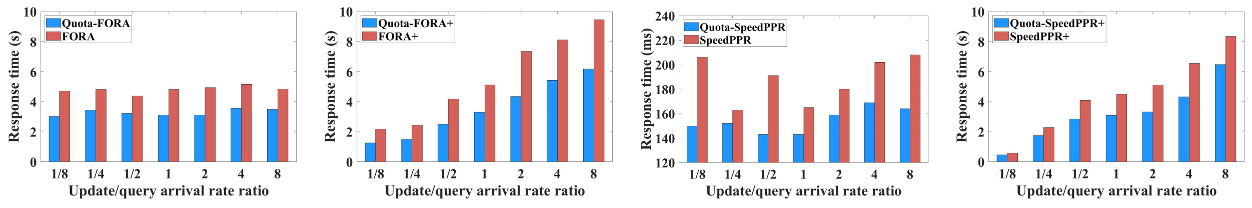


Fig. 4. Response time and empirical absolute error of *Quota* and *Agenda* with dynamic workloads. *Quota-c* is the *Quota* version that neglects hidden constants.

(3) Robustness to the arrival patterns of requests. In order to further evaluate the performance of *Quota* in evolving scenarios, we consider the situation where the queries and updates arrive at our system following multiple patterns. We relax the request arrival setting based on Poisson distribution and utilize the Uniform distribution, Geometric distribution, Normal distribution and Gamma distribution to formulate the arrival time. We trigger the queries and updates based on *LJ* datasets with the rate $\lambda_q = \lambda_u = 0.1$ and $T = 1000s$ utilizing these distributions. Moreover, we extract the true event stream log from Wikipedia [72] and examine the utility of *Quota* in real application scenarios. Note that we monitor the request arrivals and obtain the real-time λ_q and λ_u in this case. As shown in Table III, the response time of *Agenda* is sensitive to the arrival time distribution due to the crowded queue. Meanwhile, we verify the *Quota* does not degrade performance on the various kinds of workload sequences and achieves the response time deduction ranging from $(6.90 - 5.19)/6.90 = 24\%$ to $(203.78.52 - 17.96)/203.78 = 91.19\%$. This indicates that *Quota* still performs effectively in a variety of workloads.



(a) *FORA* (b) *FORA+* (c) *SpeedPPR* (d) *SpeedPPR+*

Fig. 5. Optimization results on *FORA* and *SpeedPPR*.

TABLE III
RESPONSE TIME (SECS) UNDER DIFFERENT ARRIVAL PATTERNS.

Method	Uniform distribution	Geometric distribution	Normal distribution	Gamma distribution	Wikipedia Stream
Agenda	203.78	161.22	217.52	122.57	6.90
Quota	17.96	15.68	26.64	19.99	5.19

TABLE IV
TIME COST OF CONFIGURATION.

Datasets	Grid Search	Random Search	Bayesian Optimization	Quota
Webs	96.48	56.84	25.68	0.07
DBLP	2893.83	3054.14	1968.24	0.09
LJ	165638.74	148058.36	96852.21	0.07
Twitter	-	-	-	0.08

E. Efficiency of Constrained Optimization

To better understand the configuration process of *Quota*, we give a 3-D visualization of the auto-configuration process performed by *Quota*. As an example, we show the hyperparameter search space and the corresponding response time caused by the configuration result when performing *Agenda* on *Pokec* and $\lambda_q = 0.1, \lambda_u/\lambda_q = \{1/4, 1/2, 1, 2\}$ in Figure 6 (a), (b), (c), (d) respectively. For a better comparison, we use the ratio of the tuned hyperparameter r_{max} (resp. r^b_{max}) over the default setting of $\bar{r}_{max} = \frac{1}{\alpha K}$(resp. $\bar{r}^b_{max} = \frac{1}{n}$) to demonstrate the optimization result. The red point marked "Original setting of *Agenda*" shows the situation where r_{max} and r^b_{max} equal to the default setting as $r_{max}/\bar{r}_{max} = 1$, $r^b_{max}/\bar{r}^b_{max} = 1$. The blue point marked "Selected configuration of *Quota*" represents the configuration result computed by *Quota*. We see that the default setting of *Agenda* cannot be the optimal configuration to achieve a low query response time for different workloads. Nevertheless, *Quota* conducts the constrained optimization based on the hyperparameter search space and successfully chooses the configuration with the lowest response time (e.g. $r_{max}/\bar{r}_{max} = 0.65, r^b_{max}/\bar{r}^b_{max} = 0.03$ in the (b) case).

To further illustrate *Quota*'s efficiency, we investigate the time cost of the general hyperparameter search algorithms such as Grid Search, Random Search and Bayesian Optimization mentioned in Section VII. When we employ an incomplete search space such as $r_{max}, r^b_{max} = \{0.1, 0.2, ..., 1\}$, these methods need to first pick up a configuration and test the final response time output. As shown in Table IV, Grid Search and Random Search may take up to 80 runs of *Agenda* to achieve the comparable performance as *Quota-Agenda* does, where each run consumes much time ranging from $1.44s$ for *Webs* to $16368s$ for *Twitter*. However, *Quota* only needs less than $0.1s$ to achieve the optimal solution even when the workloads change. The rationale behind this remarkable difference is that *Quota* incorporates the intrinsic time cost information of the target algorithms and has no need to depend on the computed result (i.e., response time). Correspondingly, once the workload changes, *Quota* will directly seek the optimal hyperparameter automatically within a short period of time, which again underscores the importance of an adaptive and configuration framework.

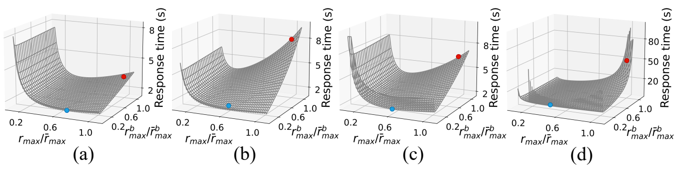


Fig. 6. *Quota* configuration.

F. Experiments on FORA and SpeedPPR

We further demonstrate the generality of *Quota* designs, and report the experimental results of *Quota-FORA*, *Quota-FORA+*, *Quota-SpeedPPR*, and *Quota-SpeedPPR+*, which respectively build *Quota* upon *FORA*, *FORA+*, *SpeedPPR*, and *SpeedPPR+* based on our designs in Section IV. Here we generate the queries and updates for *FORA*, *FORA+*, *SpeedPPR*, and *SpeedPPR+* in the same manner as *Agenda*.

Following the experimental setting of *Quota-Agenda*, we conduct the optimization processes of *FORA*, *FORA+*, *SpeedPPR*, and *SpeedPPR+* on the representative dataset *DBLP*. The optimization results after applying *Quota* on *FORA* and *SpeedPPR* are summarized in Figure 5. It can be seen that when combined with *Quota*, all four algorithms can significantly reduce the response time, regardless of whether the algorithm is index-free or index-based and the corresponding ratio between query and update arrival rates.

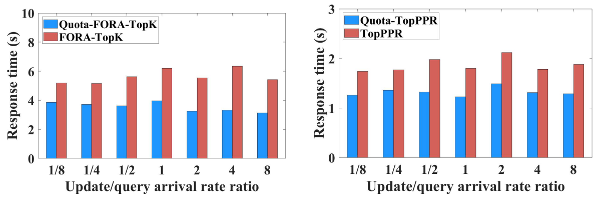
We refer to the optimization results of index-free *FORA* and index-based *FORA* + for a more detailed discussion. For index-free *FORA*, since the update process can be finished by adding or deleting an edge using $O(1)$ time, *Quota* automatically detects that the key to optimizing query response time is to minimize the query time. From Figure 5 (a) we observe that the default value of $r_{max} = \frac{1}{\sqrt{mK}}$ used in *FORA* may not always be the optimal setting to achieve the most efficient query process, where the hyperparameter r_{max} can be further tuned to improve the query efficiency. *Quota* shows a promising tuning result that it improves the query efficiency of index-free *FORA* by 25% on average across all the cases we tested. A similar situation happens for the index-based version *FORA+*, where *Quota* achieves a more pronounced improvement by up to 40%. The improvement derives from two perspectives.

First, the theoretical optimal solution $r_{max} = \frac{1}{\sqrt{mK}}$ neglects the hidden time constant as well as the practical time cost of each sub-process (e.g., *Forward Push* and *Random Walk*). By adding the hidden constant factors into modeling, *Quota* conducts a more accurate estimation for the mean query and update costs, enabling a more efficient optimization of the query response time. Second, the default setting of r_{max} may not be robust to different workloads when we test the index-based *FORA+* algorithm. This is expected because as the workload becomes update-heavy, even though *FORA+* can efficiently complete a query using only $0.1s$ on average, its update cost (particularly, index refreshes) is significant. As a result, the query response is severely delayed, reaching up to $9.46s$ (when $\lambda_q = 10$ and $\lambda_u = 80$). Fortunately, *Quota* incorporates the query and update arrival rates into the target function for optimization, effectively compromising this effect and enabling the system to adapt to various workloads.

Except for *FORA* and *FORA+*, *Quota* also improves *SpeedPPR* and *SpeedPPR+* by up to 27% and 34%. Based on the above experimental results, we conclude that *Quota* has the remarkable potential to be extended to other state-of-the-art algorithms and significantly optimizes the response time for a wide spectrum of workloads.

G. Quota on Top-k PPR

Top-k SSPPR is a PPR query type that only returns the approximate top-k highest PPR scores with respect to the source node, which is used in some applications [73], [74]. We also incorporate two state-of-the-art top-k PPR methods *FORA-TopK* [13] and TopPPR [17] in a similar fashion (namely, tuning their internal parameter r_{max}), and evaluate the effect brought by *Quota*. The results on *LJ* datasets follow the same setting mentioned above. As is shown in Figure 7, compared with the default setting of two fundamental algorithms, *Quota* can adapt to multiple query and update arrival rates, giving us up to 50% and 33% improvement of the response time for *FORA-TopK* and TopPPR, respectively. Our results hint that the original default settings in both methods are not ideal for QoS optimization and thus leave significant room for improvement.



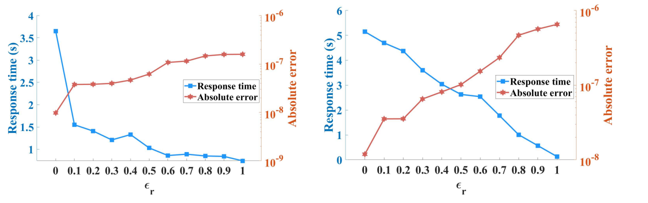
(a) *FORA-TopK* (b) *TopPPR*

Fig. 7. Optimization results on *FORA-TopK* and *TopPPR*.

H. Impact on the Reordering Algorithm

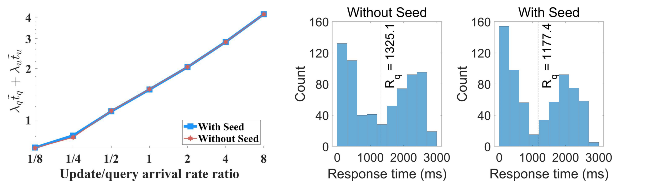
We perform an effectiveness analysis of *Seed* by setting various values for ϵ_r and show the impact on *Quota*'s performance. As mentioned in Section VI, *Seed* enables *Quota* to achieve a balance between the response time and the overall accuracy of queries. To verify this point, we further investigate how ϵ_r can impact the response time and accuracy. Particularly, we build *Quota* upon two index-based algorithms *Agenda* and *FORA+* again when setting different threshold $\epsilon_r = \{0, 0.1, 0.2, ..., 1\}$. We select an update-heavy workload ($\lambda_q = 10, \lambda_u = 40$) from the workload settings on *DBLP* datasets. As shown in Figure 8, with the increase of ϵ_r, the response time will gradually decrease, which implies the queries have a higher priority and can be finished faster. Meanwhile, we calculate the true PPR error by comparing the PPR output by *Quota* with the ground-truth PPR values. Interestingly, the true error is significantly lower than that of the theoretical guarantees, demonstrating the practical value of the reordering algorithms in reducing the response time.

Moreover, to support our claim in Lemma 3, we conduct the experiments on the overall time cost when applying various query and update rates on *Webs* datasets following our aforementioned settings with $\epsilon_r = 0.5$. Fig. 9 (a) shows that applying *Seed* has negligible impact on the overall performance $\lambda_q \tilde{t}_q + \lambda_u \tilde{t}_u$. In contrast, the response time distribution before and after applying *Seed* (($\lambda_q = 100, \lambda_u = 100$)) has been shifted. Fig. 9 (b) shows that *Seed* offsets the response time distribution and there is a larger portion of queries acquiring the short response time (e.g., below 1000ms), and this naturally gives a lower average response time.



(a) *Quota-Agenda* (b) *Quota-FORA+*

Fig. 8. Response time and true absolute error when setting different ϵ_r.



(a) *Overall performance* (b) *Response time distribution*

Fig. 9. The comparison of overall performance and response time distribution after applying *Seed*.

IX. Conclusion

This paper makes the first attempt to optimize Quality-of-Service-Aware PPR computation, which aims to minimize the query response time for PPR queries on dynamically evolving graphs. We present *Quota*, which is an auto-configuration system that converts the state-of-the-art PPR algorithms to counterparts that optimize the query response time. *Quota* incorporates several interesting mathematical tools including queuing theory, complexity analysis, and constrained optimization. Ultimately, *Quota* achieves a successful auto-configuration under various query and update workloads.

Acknowledgment

This research is supported by Singapore MOE funding (RS05/21, MOE-T2EP20122-0003) and NTU-NAP startup grant (022029-00001).

References

[1] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh, "Wtf: The who to follow service at twitter," in *Proceedings of the 22nd international conference on World Wide Web*, 2013, pp. 505–514.

[2] S. Luo, X. Xiao, W. Lin, and B. Kao, "Baton: Batch one-hop personalized pageranks with efficiency and accuracy," *IEEE Transactions on Knowledge and Data Engineering*, vol. 32, no. 10, pp. 1897–1908, 2019.

[3] 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 *Proceedings of the 26th international conference on world wide web companion*, 2017, pp. 583–592.

[4] D. Mo and S. Luo, "Single-source personalized pageranks with workload robustness," *IEEE Transactions on Knowledge and Data Engineering*, 2022.

[5] 2022. [Online]. Available: https://99firms.com/blog/pinterest-statistics/#gref.

[6] https://webtribunal.net/blog/tencent-stats/#gref, May 2022.

[7] 2022. [Online]. Available: https://www.ibm.com/docs/en/spectrum-lsf/10.1.0?topic=scheduling-understand-fairshare.

[8] J. Feng, "Fair no-unnecessary-waiting-time-fcfs allocation rule in multi-item inventory systems," *IISE Transactions*, vol. 50, no. 6, pp. 525–534, 2018.

[9] I. Grosof, K. Yang, Z. Scully, and M. Harchol-Balter, "Nudge: Stochastically improving upon fcfs," in *Abstract Proceedings of the 2021 ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, 2021, pp. 11–12.

[10] B. Bahmani, A. Chowdhury, and A. Goel, "Fast incremental and personalized pagerank," *Proceedings of the VLDB Endowment*, vol. 4, no. 3, 2010.

[11] H. Zhang, P. Lofgren, and A. Goel, "Approximate personalized pagerank on dynamic graphs," in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, 2016, pp. 1315–1324.

[12] Z. Zhan, R. Hu, X. Gao, and N. Huai, "Fast incremental pagerank on dynamic networks," in *International Conference on Web Engineering*. Springer, 2019, pp. 154–168.

[13] S. Wang, R. Yang, X. Xiao, Z. Wei, and Y. Yang, "Fora: simple and effective approximate single-source personalized pagerank," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017, pp. 505–514.

[14] P. Lofgren, S. Banerjee, and A. Goel, "Bidirectional pagerank estimation: From average-case to worst-case," in *International Workshop on Algorithms and Models for the Web-Graph*. Springer, 2015, pp. 164–176.

[15] H. Wang, Z. Wei, J. Gan, Y. Yuan, X. Du, and J.-R. Wen, "Edge-based local push for personalized pagerank," *arXiv preprint arXiv:2203.07937*, 2022.

[16] H. Wu, J. Gan, Z. Wei, and R. Zhang, "Unifying the global and local approaches: an efficient power iteration with forward push," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 1996–2008.

[17] Z. Wei, X. He, X. Xiao, S. Wang, S. Shang, and J.-R. Wen, "Topppr: top-k personalized pagerank queries with precision guarantees on large graphs," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 441–456.

[18] W. Fan, C. Tian, R. Xu, Q. Yin, W. Yu, and J. Zhou, "Incrementalizing graph algorithms," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 459–471.

[19] N. Ohsaka, T. Maehara, and K.-i. Kawarabayashi, "Efficient pagerank tracking in evolving networks," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2015, pp. 875–884.

[20] M. Yoon, W. Jin, and U. Kang, "Fast and accurate random walk with restart on dynamic graphs with guarantees," in *Proceedings of the 2018 World Wide Web Conference*, 2018, pp. 409–418.

[21] X. Guo, B. Zhou, and S. Skiena, "Subset node anomaly tracking over large dynamic graphs," in *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2022, pp. 475–485.

[22] Y. Zheng, H. Wang, Z. Wei, J. Liu, and S. Wang, "Instant graph neural networks for dynamic graphs," in *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2022, pp. 2605–2615.

[23] W. Guo, Y. Li, M. Sha, and K.-L. Tan, "Parallel personalized pagerank on dynamic graphs," *Proceedings of the VLDB Endowment*, vol. 11, no. 1, pp. 93–106, 2017.

[24] D. Eppstein, Z. Galil, and G. F. Italiano, "Dynamic graph algorithms," *Algorithms and theory of computation handbook*, vol. 1, pp. 9–1, 1999.

[25] P. Berkhin, "Bookmark-coloring algorithm for personalized pagerank computing," *Internet Mathematics*, vol. 3, no. 1, pp. 41–62, 2006.

[26] R. Andersen, F. Chung, and K. Lang, "Local graph partitioning using pagerank vectors," in *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*. IEEE, 2006, pp. 475–486.

[27] https://sites.google.com/view/quota-technical-report/, 2023.

[28] R. Andersen, C. Borgs, J. Chayes, J. Hopcraft, V. S. Mirrokni, and S.-H. Teng, "Local computation of pagerank contributions," in *Algorithms and Models for the Web-Graph: 5th International Workshop, WAW 2007, San Diego, CA, USA, December 11-12, 2007. Proceedings 5*. Springer, 2007, pp. 150–165.

[29] D. Lin, R. C.-W. Wong, M. Xie, and V. J. Wei, "Index-free approach with theoretical guarantee for efficient random walk with restart query," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 913–924.

[30] P. Berkhin, "A survey on pagerank computing," *Internet mathematics*, vol. 2, no. 1, pp. 73–120, 2005.

[31] S. Luo, B. Kao, G. Li, J. Hu, R. Cheng, and Y. Zheng, "Toain: a throughput optimizing adaptive index for answering dynamic k nn queries on road networks," *Proceedings of the VLDB Endowment*, vol. 11, no. 5, pp. 594–606, 2018.

[32] S. Moore, "Approximating the behavior of nonstationary single-server queues," *Operations Research*, vol. 23, no. 5, pp. 1011–1032, 1975.

[33] M. Bramson, *Stability of queueing networks*. Springer, 2008.

[34] R. R. Curtin, M. Edel, R. G. Prabhu, S. Basak, Z. Lou, and C. Sanderson, "The ensmallen library for flexible numerical optimization." *J. Mach. Learn. Res.*, vol. 22, pp. 166–1, 2021.

[35] M. R. Hestenes, "Multiplier and gradient methods," *Journal of optimization theory and applications*, vol. 4, no. 5, pp. 303–320, 1969.

[36] C. Zhu, R. H. Byrd, P. Lu, and J. Nocedal, "Algorithm 778: L-bfgs-b: Fortran subroutines for large-scale bound-constrained optimization," *ACM Transactions on mathematical software (TOMS)*, vol. 23, no. 4, pp. 550–560, 1997.

[37] R. Malouf, "A comparison of algorithms for maximum entropy parameter estimation," in *COLING-02: The 6th Conference on Natural Language Learning 2002 (CoNLL-2002)*, 2002.

[38] E. G. Birgin and J. M. Martínez, *Practical augmented Lagrangian methods for constrained optimization*. SIAM, 2014.

[39] D. P. Bertsekas, "Nonlinear programming," *Journal of the Operational Research Society*, vol. 48, no. 3, pp. 334–334, 1997.

[40] D. W. Coit, A. E. Smith, and D. M. Tate, "Adaptive penalty methods for genetic optimization of constrained combinatorial problems," *INFORMS Journal on Computing*, vol. 8, no. 2, pp. 173–182, 1996.

[41] M. V. Afonso, J. M. Bioucas-Dias, and M. A. Figueiredo, "An augmented lagrangian approach to the constrained optimization formulation of imaging inverse problems," *IEEE transactions on image processing*, vol. 20, no. 3, pp. 681–695, 2010.

[42] G. Lan and R. D. Monteiro, "Iteration-complexity of first-order augmented lagrangian methods for convex programming," *Mathematical Programming*, vol. 155, no. 1, pp. 511–547, 2016.

[43] P. Liashchynskyi and P. Liashchynskyi, "Grid search, random search, genetic algorithm: a big comparison for nas," *arXiv preprint arXiv:1912.06059*, 2019.

[44] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," *Advances in neural information processing systems*, vol. 25, 2012.

[45] P. Naor, "The regulation of queue size by levying tolls," *Econometrica: journal of the Econometric Society*, pp. 15–24, 1969.

[46] W. S. Vickrey, "Congestion theory and transport investment," *The American Economic Review*, pp. 251–260, 1969.

[47] G. Newell, "Queues with time-dependent arrival rates. iii—a mild rush hour," *Journal of Applied Probability*, vol. 5, no. 3, pp. 591–606, 1968.

[48] M. Haviv, O. Kella, and Y. Kerner, "Equilibrium strategies in queues based on time or index of arrival," *Probability in the Engineering and Informational Sciences*, vol. 24, no. 1, pp. 13–25, 2010.

[49] L. Kleinrock, "Theory, volume 1, queueing systems," 1975.

[50] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer networks and ISDN systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[51] N. Ma, J. Guan, and Y. Zhao, “Bringing pagerank to the citation analysis,” *Information Processing & Management*, vol. 44, no. 2, pp. 800–810, 2008.

[52] S. Kamvar, T. Haveliwala, and G. Golub, “Adaptive methods for the computation of pagerank,” *Linear Algebra and its Applications*, vol. 386, pp. 51–65, 2004.

[53] S. D. Kamvar, T. H. Haveliwala, C. Manning, and G. H. Golub, “Exploiting the block structure of the web for computing pagerank,” 2003.

[54] S. D. Kamvar, T. H. Haveliwala, C. D. Manning, and G. H. Golub, “Extrapolation methods for accelerating pagerank computations,” in *Proceedings of the 12th international conference on World Wide Web*, 2003, pp. 261–270.

[55] 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.

[56] A. D. Sarma, S. Gollapudi, and R. Panigrahy, “Estimating pagerank on graph streams,” *Journal of the ACM (JACM)*, vol. 58, no. 3, pp. 1–19, 2011.

[57] P. Lofgren, S. Banerjee, and A. Goel, “Personalized pagerank estimation and search: A bidirectional approach,” in *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*, 2016, pp. 163–172.

[58] S. Wang, Y. Tang, X. Xiao, Y. Yang, and Z. Li, “Hubppr: effective indexing for approximate personalized pagerank,” *Proceedings of the VLDB Endowment*, vol. 10, no. 3, pp. 205–216, 2016.

[59] J. Shi, R. Yang, T. Jin, X. Xiao, and Y. Yang, “Realtime top-k personalized pagerank over large graphs on gpus,” *Proceedings of the VLDB Endowment*, vol. 13, no. 1, pp. 15–28, 2019.

[60] H. Wang, Z. Wei, J. Gan, S. Wang, and Z. Huang, “Personalized pagerank to a target node, revisited,” in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 657–667.

[61] P. A. Lofgren, S. Banerjee, A. Goel, and C. Seshadhri, “Fast-ppr: Scaling personalized pagerank estimation for large graphs,” in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014, pp. 1436–1445.

[62] G. Hou, X. Chen, S. Wang, and Z. Wei, “Massively parallel algorithms for personalized pagerank,” *Proceedings of the VLDB Endowment*, vol. 14, no. 9, pp. 1668–1680, 2021.

[63] A. Das Sarma, A. R. Molla, G. Pandurangan, and E. Upfal, “Fast distributed pagerank computation,” in *International Conference on Distributed Computing and Networking*, 2013, pp. 11–26.

[64] S. Luo, “Improved communication cost in distributed pagerank computation–a theoretical study,” in *International Conference on Machine Learning*. PMLR, 2020, pp. 6459–6467.

[65] W. Lin, “Distributed algorithms for fully personalized pagerank on large graphs,” in *The World Wide Web Conference*, 2019, pp. 1084–1094.

[66] R. Wang, S. Wang, and X. Zhou, “Parallelizing approximate single-source personalized pagerank queries on shared memory,” *The VLDB Journal*, vol. 28, no. 6, pp. 923–940, 2019.

[67] W. Yu and J. McCann, “Random walk with restart over dynamic graphs,” in *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE, 2016, pp. 589–598.

[68] S. Chakrabarti, “Dynamic personalized pagerank in entity-relation graphs,” in *Proceedings of the 16th international conference on World Wide Web*, 2007, pp. 571–580.

[69] T. Maehara, T. Akiba, Y. Iwata, and K.-i. Kawarabayashi, “Computing personalized pagerank quickly by exploiting graph structures,” *Proceedings of the VLDB Endowment*, vol. 7, no. 12, pp. 1023–1034, 2014.

[70] Q. Liu, Z. Li, J. C. Lui, and J. Cheng, “Powerwalk: Scalable personalized pagerank via random walks with vertex-centric decomposition,” in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, 2016, pp. 195–204.

[71] M. Liao, R.-H. Li, Q. Dai, and G. Wang, “Efficient personalized pagerank computation: A spanning forests sampling based approach,” in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 2048–2061.

[72] https://stream.wikimedia.org/v2/ui/#/, May 2023.

[73] Y. Wu, R. Jin, and X. Zhang, “Fast and unified local search for random walk based k-nearest-neighbor query in large graphs,” in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014, pp. 1139–1150.

[74] M. Coskun, A. Grama, and M. Koyuturk, “Efficient processing of network proximity queries via chebyshev acceleration,” in *Proceedings of the 22nd ACM SIGKDD International conference on knowledge discovery and data mining*, 2016, pp. 1515–1524.

[75] https://sites.google.com/view/agenda-technical-report/, 2020.

[76] M. Sipser, “Introduction to the theory of computation,” *ACM Sigact News*, vol. 27, no. 1, pp. 27–29, 1996.

[77] J. Leskovec, D. Huttenlocher, and J. Kleinberg, “Signed networks in social media,” in *Proceedings of the SIGCHI conference on human factors in computing systems*, 2010, pp. 1361–1370.

[78] W. Whitt, “Heavy traffic limit theorems for queues: a survey,” in *Mathematical Methods in Queueing Theory*. Springer, 1974, pp. 307–350.

[79] W. A. Massey and W. Whitt, “Unstable asymptotics for nonstationary queues,” *Mathematics of Operations Research*, vol. 19, no. 2, pp. 267–291, 1994.

Appendix

A. Frequently used notations and Forward Push algorithm

TABLE V
FREQUENTLY USED NOTATIONS IN THIS PAPER.

Notations	Descriptions
$G_i(V_i, E_i)$	Directed graph after i edge updates
n_i, m_i	$n_i = \lvert V_i\rvert, m_i = \lvert E_i\rvert$
n, m	Numbers of nodes and edges in the current graph.
$\pi(G_i, s, t)$	PPR of node t from the source node s in graph G_i
$r(G_i, s, t)$	Residue of node t from the source node s in G_i
$\pi^\circ(G_i, s, t)$	Reserve of node t from source node s in graph G_i
$d^{out}(G_i, t)$	out-degree of node t in graph G_i
ϵ	PPR accuracy guarantee
p_f	Failure probability of accuracy guarantee
δ	PPR threshold
α	Teleport probability of random walks
λ_q, λ_u	Query/update arrival rate
R_q	Average response time
V_q, V_u	Variance of the query/update time
$\tilde{t}_q, \tilde{t}_u$	Average query/update time
r_{max}, r^b_{max}	Threshold in *Forward Push* and *Reverse Push*
τ	Time constant

Algorithm 3: Forward Push

```
Input : Graph $G = (V, E)$, teleport probability $\alpha$, source node $s$, residue threshold $r_{max}$
Output: Reserve $\pi^\circ(G, s, t)$ and residue $r(G, s, t)$ for each $t \in V$
1 Initialize residue $r(G, s, s) = 1$; $r(G, s, t) = 0$ and reserve $\pi^\circ(G, s, t) = 0$ for each $t \in V$;
2 while $\exists t \in V$ s.t. $\frac{r(G,s,t)}{d^{out}(G,t)} > r_{max}$ do
3     foreach $v$ that is an out-neighbor of $t$ do
4         $r(G, s, v) += (1 - \alpha) \cdot \frac{r(G,s,t)}{d^{out}(G,t)}$;
5     $\pi^\circ(G, s, t) += \alpha \cdot r(G, s, t)$;
6     $r(G, s, t) = 0$;
```

B. Incorporating Agenda

In this section, we show how to integrate *Quota* on *Agenda* [4], a recent state-of-the-art dynamic PPR algorithm. We give refined analysis that is particularly useful for the integration.

1) Complexity of Agenda: In order to model the mean query and update time, we first conduct an in-depth analysis of its query and update costs. As mentioned in Section III-B, *Agenda* applies the *Push+Walk* framework on the query and conducts the random walk index update only when needed. The query and update algorithms in *Agenda* can be wrapped up using a few sub-processes as follows.

- **Query:** *Forward Push*, *Lazy Index Update*, *Random Walk*.
- **Update:** *Reverse Push*, *Index Inaccuracy Update*.

Splitting the query or update into sub-processes facilitates the later cost modeling. Next, we provide the analysis for more details of the query and update process in *Agenda*, in order to incorporate *Agenda* into *Quota*.

TABLE VI
COST FUNCTIONS OF *Agenda* SUB-PROCESSES.

	Sub-processes	Time Cost Function
Query	*Forward Push*	$\frac{1}{r_{max}} \cdot \tau_1$
	Lazy Index Update	$\frac{\lambda_u r_{max}(nr^b_{\max}+1)}{\lambda_q} \cdot \tau_2$
	Random Walk	$r_{max} \cdot \tau_3$
Update	*Reverse Push*	$\frac{1}{r^b_{max}} \cdot \tau_4$
	Index Inaccuracy Update	$1 \cdot \tau_5$

Query-*<u>Forward Push</u>*. By Lemma 13 in [11], *Forward Push* in Algorithm 3 has a complexity of $O(\frac{1}{\alpha r_{max}})$, where r_{max} is the threshold for Forward Push (see Line 2 of Algorithm 3).

Query-*<u>Lazy Index Update</u>*. Given $K = \frac{(2\epsilon/3+2)\log(1/p_f)}{\epsilon^2\delta}$, we have the following lemma [75]:

Lemma 4. *An edge insert or delete in Agenda will trigger at most* $\frac{2mr_{max}(nr^b_{\max}+1)}{\epsilon\delta n\alpha}$ *fraction of the overall index to be updated.*

In dynamic PPR scenarios, we assume that there occur λ_q queries and λ_u updates per second. Therefore the overall faction of the index that needs to be updated due to these λ_u updates per second is at most $O\left(\frac{2\lambda_u m r_{max}(nr^b_{\max}+1)}{\epsilon\delta n\alpha}\right)$. Finally, we can derive the amortized index update fraction for each query as $O\left(\frac{2\lambda_u r_{max} m(nr^b_{\max}+1)}{\lambda_q\epsilon\delta n\alpha}\right)$.

Query-*<u>Random Walk</u>*. In the *Random Walk* sub-process, there are $r(G_i, s, t) \cdot K$ random walks sampled for each residue value $r(G_i, s, t)$ where $t \in V_i$. The total number of random walks sampled is $\sum_{t\in V_i} r(G_i, s, t)\cdot K$. Since the residue value is computed by the *Forward Push* sub-process, we have:

$$\sum_{t\in V_i} r(G_i, s, t) \cdot K \le nr_{\max}K$$

The number of random walks sampled during the *Random Walk* sub-process is $O(nr_{max}K)$.

Update-*<u>Reverse Push</u>*. We adopt the conclusion of Theorem 1 in Fast-PPR [61] to derive the complexity of *Reverse Push*. That is, given the reverse threshold r^b_{max} and the graph average degree $\bar{d} = \frac{m}{n}$, the complexity of *Reverse Push* is $O\left(\frac{\bar{d}}{\alpha r^b_{max}}\right) = O\left(\frac{m}{\alpha n r^b_{max}}\right)$.

Update-*<u>Index Inaccuracy Update</u>*. During the last sub-process of update, *Index Inaccuracy Update* computes the inaccuracy value for any node $t \in V_i$. Hence, the complexity of *Index Inaccuracy Update* is $O(n)$.

2) Query and Update Cost Functions: With the above analysis, we can express the practical computation cost based on the hyperparameters in *Agenda* and some constant factors. Particularly, the amount of time consumed by the algorithms is related to the time complexity expression and the hidden constant factors [76]. Taking the *Forward Push* as an example, we can quantify the cost of *Forward Push* as $\frac{1}{\alpha r_{max}} \cdot \tau_1$, where $\frac{1}{\alpha r_{max}}$ refers to the corresponding complexity expression for *Forward Push* and τ_1 refers to the unit time cost of changing a node's residue or reserve value.

Without loss of generality, we target two important tunable hyperparameters r_{max} and r^b_{max} ($0 < r_{max}, r^b_{max} < 1$), which are the thresholds in the *Forward Push* and *Reverse Push* sub-process. Given $\boldsymbol{\beta} = (r_{max}, r^b_{max})$, we note that tuning these two hyperparameters will still maintain the worst-case accuracy guarantee with respect to the estimated PPR values. We simplify the time complexity and the extra coefficients (e.g., α, δ, m *etc.*) are reflected in the time constant τ. Based on the cost of each sub-process summarized in Table VI, we can express $\tilde{t}_q$ and $\tilde{t}_u$ as:

$$\tilde{t}_q(r_{max}, r^b_{max}) = \underbrace{\frac{1}{r_{max}} \cdot \tau_1}_{\textit{Forward Push}} + \underbrace{\frac{\lambda_u r_{max}(n r^b_{\max} + 1)}{\lambda_q} \cdot \tau_2}_{\textit{Lazy Index Update}} + \underbrace{r_{max} \cdot \tau_3}_{\textit{Random Walk}}, \quad \tilde{t}_u(r_{max}, r^b_{max}) = \underbrace{\frac{1}{r^b_{max}} \cdot \tau_4}_{\textit{Reverse Push}} + \underbrace{1 \cdot \tau_5}_{\textit{Index Update}},$$

where $\tau_i (1 \leq i \leq 5)$ denotes the different constant factors. For the reason that τ_i incorporates the coefficients in the query and update time complexity, we note that τ_i depends on the experiment setting and will be measured during our implementation. The formulated mean query and update cost functions can be integrated into the optimization process in Section IV-A, and utilized to compute the optimal settings for r_{max} and r^b_{max}.

3) Constraint Functions: Last, we present the constraint functions designed for *Agenda* to finalize the target function $\Phi(\boldsymbol{\beta})$ in Eq. 6. Given the hyperparameter vector $\boldsymbol{\beta} = (r_{max}, r^b_{max})$, we need to ensure that the hyperparameters are searched in the hyperparameter space Λ, where we have $0 < r_{max} < 1$ and $0 < r^b_{max} < 1$. In order to meet the constraint requirements, the constraint functions are designed as follows:

$$C_1(r_{max}, r^b_{max}) = \lambda_q \tilde{t}_q(r_{max}, r^b_{max}) + \lambda_u \tilde{t}_u(r_{max}, r^b_{max}) - 1,$$
$$C_2(r_{max}, r^b_{max}) = -r_{max}, \qquad C_3(r_{max}, r^b_{max}) = r_{max} - 1,$$
$$C_4(r_{max}, r^b_{max}) = -r^b_{max}, \qquad C_5(r_{max}, r^b_{max}) = r^b_{max} - 1,$$

where $C_1(r_{max}, r^b_{max})$ represents the stability constraint and other functions are due to the search space constraint.

C. Incorporating FORA and SpeedPPR

In the previous section, we presented the optimization process of *Quota* using the state-of-the-art algorithm *Agenda*. However, it is worth noting that *Quota* can be easily integrated with other PPR algorithms that include query and update processes. In this section, we demonstrate how to incorporate *Quota* into two other representative state-of-the-art algorithms that use similar *Push+Walk* structures: *FORA* [13] and *SpeedPPR* [16]. For completeness, we also provide a complexity analysis of index-free *FORA*, *SpeedPPR*, index-based *FORA+*, and *SpeedPPR+* as follows.

For each query, as given in [13], both *FORA* and *FORA+* consume $O\left(\frac{1}{\alpha r_{\max}} + m r_{max} K\right)$ time, where $K = (2\epsilon/3 + 2)\log(2/p_f)/(\epsilon^2\delta)$. For the index-free *FORA*, the update process is only conducted by adding or deleting an edge on the graph, which costs $O(1)$ time only. For *FORA+* that does not provide an update function, we achieve the update operation by regenerating the index. This consumes $O(m r_{max} K)$ time because there requires to generate such number of random walks. Similarly, as shown in [16], for both *SpeedPPR+* and *SpeedPPR*, each query consumes $O(m \log \frac{1}{r_{max} m} + m r_{max} W)$ time, where $W = 2(2\epsilon/3 + 2)\log n/(\epsilon^2\delta)$. Moreover, each update will take $O(1)$ time in the index-free method *SpeedPPR*, but $O(m r_{max} W)$ to reconstruct the index for *SpeedPPR+*.

We note that while the query costs for *FORA* and *FORA+* have the same form, they entail different empirical performance. To explain, in the random walk sub-process during a query, *FORA* conducts random walks online, whereas *FORA+*, being an index-based method, directly extracts the random walk results from the index. Consequently, *FORA* and *FORA+* entail different constant coefficients. A similar analysis also applies to *SpeedPPR* and *SpeedPPR+*.

For the index-free *FORA* and *SpeedPPR*, the update process maintains $O(1)$ time cost and the response time optimization is equally transformed into optimizing query time. Although *FORA* [13] attempts to achieve the theoretical minimization of query time complexity $O\left(1/(\alpha r_{\max}) + m r_{max} K\right)$ by setting the default hyperparameter as $r_{max} = 1/\sqrt{\alpha m K}$ where $K = (2\epsilon/3 + 2)\log(2/p_f)/(\epsilon^2\delta)$, our experimental results in Section VIII prove that the default setting may not be the optimal solution for minimizing the query time. On the contrary, *Quota* takes the practical time cost of each sub-process of the target algorithm into consideration and fundamentally improves the query efficiency.

Employing *Quota* on *FORA* and *SpeedPPR* is relatively simple since there only exists a tunable hyperparameter r_{max} which has no effect on the accuracy guarantees. After considering different time constants, the mean query and update time function in *FORA*, *FORA+*, *SpeedPPR*, and *SpeedPPR+* can be formulated in Table I (bottom).

- *FORA*. After considering the hyperparameter r_{max} and the time constant τ_1 and τ_2 corresponding to *Forward Push* and *Random Walk* sub-processes, we can formulate the mean query and update time of *FORA* as:

$$\tilde{t}_q(r_{max}) = \frac{1}{r_{\max}} \cdot \tau_1 + r_{max} \cdot \tau_2, \quad \tilde{t}_u(r_{max}) = 1 \cdot \tau_3,$$

where the equation of $\tilde{t}_u(r_{max})$ holds because the index-free version of *FORA* is updated by only adding or deleting an edge.

- *FORA+*. The index-based *FORA+* differs from the index-free *FORA* in that *FORA+* needs to update the random walk index when there comes an edge insert or delete, incurring:

$$\tilde{t}_q(r_{max}) = \frac{1}{r_{\max}} \cdot \tau_1 + r_{max} \cdot \tau_2, \quad \tilde{t}_u(r_{max}) = r_{max} \cdot \tau_3$$

TABLE VII
RESPONSE TIME (MILLISECS) UNDER EXTREME SITUATIONS.

	$\lambda_q = 10$ $\lambda_u = 10$	$\lambda_q = 10$ $\lambda_u = 20$	$\lambda_q = 10$ $\lambda_u = 40$	$\lambda_q = 200$ $\lambda_u = 200$	$\lambda_q = 200$ $\lambda_u = 400$	$\lambda_q = 200$ $\lambda_u = 800$
Agenda	16.68	19.53	31.17	3477.76	4105.62	4379.24
Quota	7.69	8.13	16.44	3064.76	3638.41	4072.92

The query and update cost functions for *SpeedPPR* and *SpeedPPR+* can be derived in a similar fashion as discussed before. To complete the optimization process for *FORA* and *SpeedPPR*, we replace the mean query and update functions in Eq. 2 with the derived equations. Then, we formulate the objective function for *FORA*, *FORA+*, *SpeedPPR*, and *SpeedPPR+* and perform the constrained optimization process similar to that of *Agenda*.

In conclusion, *Quota* can be easily applied to these two representative algorithms, which highlights the versatility of the *Quota* framework. Extensive experiments on these algorithms are presented in Section VIII.

D. Response time under extreme situations

Expect for the extremely workload-heavy situations mentioned above, we additionally simulate several extremely workload-light and workload-heavy situation and assess the robustness of *Quota*. The corresponding experimental results in Table VII evaluate the performance of *Quota* and *Agenda* on the *Webs* datasets under specific query and update rate settings. We note that even in these cases, *Quota-Agenda* still achieves comparable performance, demonstrating its robustness.

E. Details of applying Wikipedia datasets

Wikipedia datasets are from [77] and we extract 100 events including page queries and changes from the event data of [72]. We record the source node, request type, and arrival time stamp to simulate the real workload. We repeat this process for 10 times and then record the mean page query response time.

F. More results under dynamic situations

In order to prove the effectiveness of *Quota* under dynamic situations, we provide more results on *DBLP* following the setting demonstrated in Figure 4, while shuffling the order of various patterns and verify the performance in different workloads. The results in Figure 10 coincide with the observation in Figure 4: *Quota* can quickly tune the hyperparameters and reduce the response time when maintaining a high level of accuracy in different evolving workloads. Moreover, we reconduct the experiments in Figure 3 but change the update rates over time. Specifically, the term λ_u/λ_q changes from $1/8$ to 8 almost every 10s on *Webs* and *DBLP* datasets (also following a Poisson distribution) and every 1000s on *Pokec* and *LJ* datasets. We continuously monitor the rates and perform the hyperparameter search every 1s on *Webs* and *DBLP* datasets and every 100s on *Pokec* and *LJ* datasets. As shown in Figure 11, *Quota* continues to effectively reduce response times over time, even when faced with evolving update rates. This further demonstrates the robustness of *Quota* in dynamic scenarios.

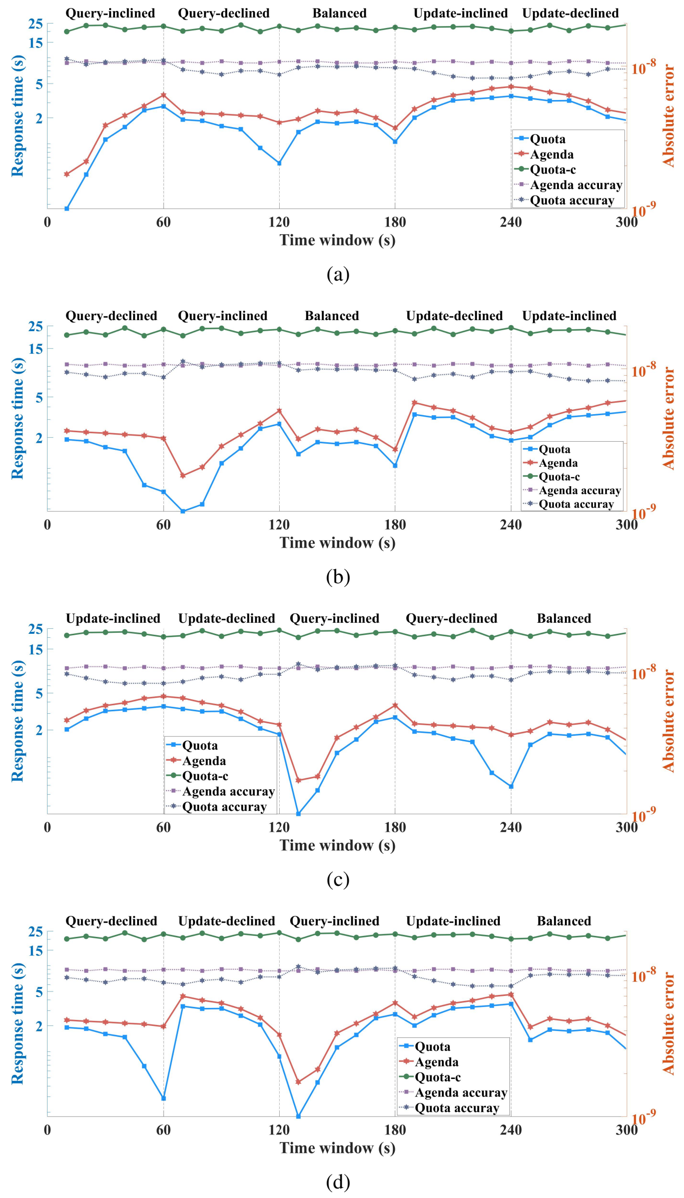


Fig. 10. More results about the response time and empirical absolute error of *Quota* and *Agenda* with dynamic workloads.

G. Cost Balance Among Sub-Processes

As analyzed in Section IV, *Quota* coordinates the queuing status by tuning the hyperparameters and maintaining the optimal trade-off between average query and update time. To further illustrate this point by experiments, we summarize the average cost of sub-processes in *Agenda* and *Quota-Agenda* on the representative tests and review how *Quota* achieves the cost balance among sub-processes. In Table VIII, we provide the optimization result when $\lambda_q = 0.1$ and $\lambda_u = \{0.05, 0.2\}$ on *LJ* datasets, respectively. From the overall performance in Figure 3 and the example in Table VIII, we observe that *Quota* can adaptively tune the average time cost of sub-processes in query and update according to query and update arrival rates. Particularly, *Quota* inherently balances the computing resource allocated for queries and updates. One can refer to the statistics for $\lambda_q = 0.1$ and $\lambda_u = 0.05$, which indicates that by the trade-

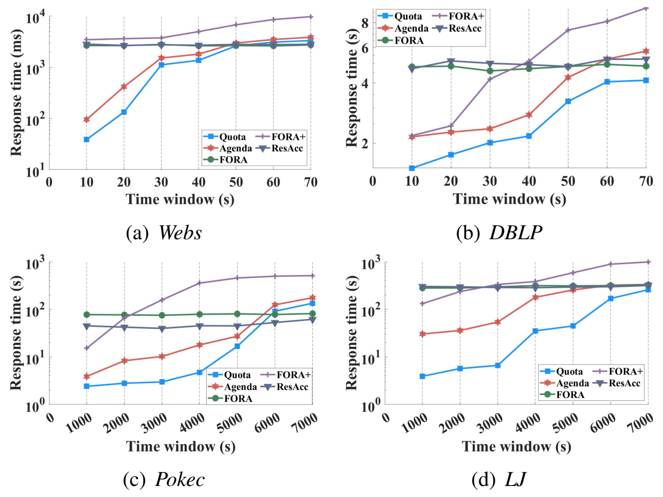


Fig. 11. More results about the response time under dynamic situations.

offs among all sub-process costs, *Quota* successfully optimizes the queuing status and ultimately reduces the response time of *Agenda* by up to $86.44\%(= (55.08 - 7.47)/55.08)$.

TABLE VIII
AVERAGE COST (SECS) WHEN $\lambda_q = 0.1$ AND $\lambda_u = \{0.05, 0.2\}$.

Sub-process	**Agenda**		**Quota-Agenda**	
	$\lambda_u = 0.05$	$\lambda_u = 0.2$	$\lambda_u = 0.05$	$\lambda_u = 0.2$
Forward Push	1.24	1.19	2.76	5.67
Lazy Index Update	7.70	16.84	0.54	1.76
Random Walk	0.73	1.03	0.71	0.88
Reverse Push	0.05	0.05	2.70	1.59
Index Inaccuracy Update	0.01	0.01	0.01	0.01
Query cost	9.67	19.06	4.01	8.31
Update cost	0.07	0.07	2.72	1.60
Response time	55.08	249.25	7.47	50.94

H. Proof of Lemma 1

By definition, the response time of N_q-th query is the difference between its arrival time A_{N_q} and departure time D_{N_q}:

$$W_{N_q} = D_{N_q} - A_{N_q} \tag{7}$$

The query arrivals follow the Poisson distribution and the arrival time A_{N_q} is calculated based on the query rate λ_q:

$$N_q^{-1} A_{N_q} = \frac{1}{\lambda_q}, \text{w.p.1 as } N_q \to \infty \tag{8}$$

Following the FCFS policy, the N_q-th query will be processed only when all of the previous queries and updates are finished. Hence for the departure time D_{N_q}, it includes three sources of time cost: (1) the query cost Q_{N_q} of these N_q queries; (2) the update cost U_{N_q} before N_q-th query; (3) the idle time ξ in the queue. Then by the linearity of expectation, we have:

$$D_{N_q} = Q_{N_q} + U_{N_q} + \xi$$

Note that $Q_{N_q} = N_q \cdot \tilde{t}_q$, and when the N_q-th ($N_q \to \infty$) query arrives, there are $(N_q/\lambda_q) \cdot \lambda_u$ updates before this query. This gives us $U_{N_q} = (N_q/\lambda_q) \cdot \lambda_u \cdot \tilde{t}_u$, and hence,

$$N_q^{-1} D_{N_q} = \tilde{t}_q + \frac{\lambda_u \tilde{t}_u}{\lambda_q} + N_q^{-1}\xi, \text{w.p.1 as } N_q \to \infty$$

Finally, based on the strong laws of large numbers in [78], [79], when $N_q \to \infty$, the idle interval between two continuous requests (query or update) in an unstable queue will converge into 0. That is, the idle time ξ will converge into a limited value and the queue will converge to the status which is totally saturated with queries and updates, which leads to:

$$N_q^{-1}\xi \equiv 0, \text{w.p.1 as } N_q \to \infty.$$

We then calculate the expected value of departure time D_{N_q}:

$$N_q^{-1} D_{N_q} \equiv \tilde{t}_q + \frac{\lambda_u \tilde{t}_u}{\lambda_q}, \text{w.p.1 as } N_q \to \infty. \tag{9}$$

We derive Eq. 7 by incorporating Eq. 8 and Eq. 9 as:

$$N_q^{-1} W_{N_q} \equiv \frac{\lambda_q \tilde{t}_q + \lambda_u \tilde{t}_u - 1}{\lambda_q}, \text{w.p.1 as } N_q \to \infty$$

I. Proof of Lemma 2

First, we need to prove a preliminary lemma:

Lemma 5. *Given any $u, v \in V_i (u \neq v)$, the following holds:*

$$\pi(G_i, u, v) \leq e(G_i, u) - \alpha. \tag{10}$$

where $e(G_i, u) = \frac{d^{out}(G_i,u) - \alpha(1-\alpha)\left(d^{out}(G_i,u) - 1\right)}{d^{out}(G_i,u)}, u, v \in V_i.$

Proof. By Lemma 1 of [2], we have $\pi(G_i, u, w) \geq \alpha(1-\alpha)/d^{out}(G_i, u)$, where w is any out-neighbor of u. And $\pi(G_i, u, u)$ is larger than α. Considering all neighbor w of u and v (v also might be one neighbor of u), we have

$$\pi(G_i, u, v) + \pi(G_i, u, u) + \sum_{w \in (Nei(u) - v)} \pi(G_i, u, w) \leq 1. \tag{11}$$

Then we can obtain the upper bound of $\pi(G_i, u, v)$ as:

$$\begin{aligned} \pi(G_i, u, v) &\leq 1 - \sum_{w \in (Nei(u) - v)} \pi(G_i, u, w) - \alpha \qquad (12) \\ &\leq 1 - \frac{\alpha(1-\alpha)}{d^{out}(G_i, u)} \cdot (d^{out}(G_i, u) - 1) - \alpha \end{aligned}$$

The proof is finished. □

We know $\pi(G_i, s, t)$ is the probability that a random walk starting from s ends at t. We express the node set passed by this random walk as $s \rightsquigarrow t$. We can divide this probability into two part:

$$\pi(G_i, s, t) = Pr(G_i, u_{i+1} \in s \rightsquigarrow t) + Pr(G_i, u_{i+1} \notin s \rightsquigarrow t) \tag{13}$$

After updating (u_{i+1}, v_{i+1}), the probability that a random walk from s not passing u_{i+1} ends at t will not change. For the term of $Pr(G_i, u_{i+1} \in s \rightsquigarrow t)$, we have:

$$Pr(G_i, u_{i+1} \in s \rightsquigarrow t) = Pr(G_i, \text{Walk from } s \text{ first passing } u_{i+1}) \cdot \pi(G_{i+1}, u_{i+1}, t). \tag{14}$$

Hence we have:

$$\begin{aligned} & |\pi(G_{i+1}, s, t) - \pi(G_i, s, t)| \\ &= |Pr(G_{i+1}, u_{i+1} \in s \rightsquigarrow t) - Pr(G_i, u_{i+1} \in s \rightsquigarrow t)| \\ &= |Pr(G_i, \text{Walk from } s \text{ first passing } u_{i+1}) \cdot \pi(G_{i+1}, u_{i+1}, t) \\ &- Pr(G_i, \text{Walk from } s \text{ first passing } u_{i+1}) \cdot \pi(G_i, u_{i+1}, t)| \\ &= Pr(G_i, \text{Walk from } s \text{ first passing } u_{i+1}) \cdot |\pi(G_{i+1}, u_{i+1}, t) \\ &- \pi(G_i, u_{i+1}, t)| \end{aligned} \tag{15}$$

The probability of a random walk from s passing u_{i+1} is $\frac{\pi(G_i,s,u_{i+1})}{\alpha}$, and the probability of a random walk from s first passing u_{i+1} is smaller than $\frac{\pi(G_i,s,u_{i+1})}{\alpha}$. Moreover, we utilize the conclusion in Theorem 1 in [4] and then we have:

$$\max_{t \in V_i} |\pi(G_{i+1}, u, t) - \pi(G_i, u, t)| \leq \frac{\pi(G_i, u, u_{i+1})}{\alpha d^{out}(G_{i+1}, u_{i+1})}. \tag{16}$$

We use u_{i+1} to replace u in Eq. 16, we have $|\pi(G_{i+1}, u_{i+1}, t) - \pi(G_i, u_{i+1}, t)| \leq \frac{\pi(G_i,u_{i+1},u_{i+1})}{\alpha d^{out}(G_{i+1},u_{i+1})}$
Hence we have:

$$\begin{aligned} & |\pi(G_{i+1}, s, t) - \pi(G_i, s, t)| \\ &\leq \frac{\pi(G_i, s, u_{i+1})}{\alpha} \cdot |\pi(G_{i+1}, u_{i+1}, t) - \pi(G_i, u_{i+1}, t)| \\ &\leq \frac{\pi(G_i, s, u_{i+1}) \pi(G_i, u_{i+1}, u_{i+1})}{\alpha^2 d^{out}(G_{i+1}, u_{i+1})} \end{aligned} \tag{17}$$

Then we uitilize the result of Lemma 5, we have $\pi(G_i, s, u_{i+1}) \leq e(G_i, s) - \alpha$, and $\pi(G_i, u_{i+1}, u_{i+1}) \leq 1 - \alpha(1-\alpha)$ and we finish the proof.

J. *Proof of Lemma 3*

We first prove that $\lambda_q \tilde{t}_q + \lambda_u \tilde{t}_u$ will remain the same after applying *Seed*. Under the Poisson distribution, we define there exist N_q queries and N_u updates in a time window T, and we know $\frac{N_q}{T} = \lambda_q$, $\frac{N_u}{T} = \lambda_u$, w.p.1 as $T \to \infty$. Then we have:

$$(N_q \tilde{t}_q + N_u \tilde{t}_u)/T = \lambda_q \tilde{t}_q + \lambda_u \tilde{t}_u, \text{w.p.1 as } T \to \infty. \tag{18}$$

Here we assume that the cost of each request will remain the same after reordering. Before applying *Seed*, the overall consumption to process all requests is $N_q \tilde{t}_q + N_u \tilde{t}_u$. After applying *Seed*, the order of these requests changes while the overall consumption is still $N_q \tilde{t}_q + N_u \tilde{t}_u$. Given a same time window T ($T \to \infty$), the term $\lambda_q \tilde{t}_q + \lambda_u \tilde{t}_u$ will not change after we apply *Seed*.

Assume the original response time of the N_q-th query before reordering is $W^*_{N_q}$. Then we prove that for the N_q-th query, the response time W_{N_q} after applying *Seed* satisfies $W_{N_q} \leq W^*_{N_q}$. Following the proof of Lemma 1, the response time $W^*_{N_q}$ can be formulated as:

$$W^*_{N_q} = D^*_{N_q} - A^*_{N_q} = Q^*_{N_q} + U^*_{N_q} + \xi^* - A^*_{N_q}, \tag{19}$$

where $Q^*_{N_q}$ is the query cost of these N_q queries, $U^*_{N_q}$ is the update cost before N_q-th query, ξ^* is the idle time, and $A^*_{N_q}$ is the arrival time of N_q-th query. Next we see the status after applying reordering and we denote the corrosponding variables as Q_{N_q}, U_{N_q}, ξ, and A_{N_q}.

To measure the response time of N_q-th query, we need to consider two situations: (a) when receiving N_q-th query, the pending update queue U^p is empty; (b) when receiving the N_q-th query, the pending update queue U^p is not empty. For (a), before processing N_q-th query, the requests before this query have been finished and $Q_{N_q} = Q^*_{N_q}$, $U_{N_q} = U^*_{N_q}$, $\xi = \xi^*$. Furthermore, the arrival time $A^*_{N_q}$ will not be affected by the reordering algorithm, and we have $A_{N_q} = A^*_{N_q}$. In this situation, we have $W_{N_q} = D_{N_q} - A_{N_q} = Q_{N_q} + U_{N_q} + \xi - A_{N_q} = W^*_{N_q}$. For (b), the only difference with (a) is that there exist updates in the pending queue U^p. Hence the time consumption U_{N_q} of processing updates will be reduced and we have $U_{N_q} \leq U^*_{N_q}$. In this situation, we have $W_{N_q} = D_{N_q} - A_{N_q} = Q_{N_q} + U_{N_q} + \xi - A_{N_q} < W^*_{N_q}$. Summarizing the results above, we have $W_{N_q} \leq W^*_{N_q}$, and the proof is finished.