Incremental and Accuracy-Aware Personalized PageRank through Scheduled Approximation*

Fanwei Zhu †, Yuan Fang #⋆, Kevin Chen-Chuan Chang #⋆, Jing Ying †

† Zhejiang University City College, China {zhufw,yingj@zucc.edu.cn}
# University of Illinois at Urbana-Champaign, USA {fang2, kcchang}@illinois.edu
⋆ Advanced Digital Sciences Center, Singapore

ABSTRACT

As Personalized PageRank has been widely leveraged for ranking on a graph, the efficient computation of Personalized PageRank Vector (PPV) becomes a prominent issue. In this paper, we propose FastPPV, an approximate PPV computation algorithm that is *incremental* and *accuracy-aware*. Our approach hinges on a novel paradigm of *scheduled approximation*: the computation is partitioned and scheduled for processing in an "organized" way, such that we can gradually improve our PPV estimation in an incremental manner, and quantify the accuracy of our approximation at query time. Guided by this principle, we develop an efficient hub based realization, where we adopt the metric of *hub-length* to partition and schedule random walk tours so that the approximation error reduces exponentially over iterations. Furthermore, as tours are segmented by hubs, the shared substructures between different tours (around the same hub) can be reused to speed up query processing both within and across iterations. Finally, we evaluate FastPPV over two real-world graphs, and show that it not only significantly outperforms two state-of-the-art baselines in both online and offline phrases, but also scale well on larger graphs. In particular, we are able to achieve near-constant time online query processing irrespective of graph size.

1. INTRODUCTION

Graphs are ubiquitous in the real-world, such as the Web, social networks and entity-relationship graphs, calling for solutions to ranking on a graph. Formally, a graph $G = (V, E)$ is represented by a set of nodes V and edges E. As each edge embeds certain semantic relationship between the nodes, given a node $q \in V$ as the query, what are the nodes *relevant* to q through the edges in E? Here, the *input* is a query q, and the *output* is a ranked list of nodes in V. We motivate such ranking with two example scenarios.

Scenario 1: Bibliographic search. Consider a bibliographic network with interconnected nodes such as papers, venues and authors. Given a paper, who are the best matching experts to review it? In this case, the input query is a paper node, and the output is a ranking over the author nodes in the network.

Scenario 2: Friends recommendation. Consider a social network with users as nodes, connected by their friendships. Given a user in the network, how can we recommend some potential friends to her? Taking the user node as the input query, a ranking over all the other user nodes can be leveraged for the recommendation.

In the above scenarios, the rankings are specific to the dynamic queries, reflecting the "relevance" of nodes to the query node. As a well-studied graph ranking algorithm, *Personalized PageRank* [14, 12] is effective in calculating such query-specific relevance based on the link structure of the graph. In this paper, we study the efficiency aspect of Personalized PageRank, as we shall see.

Background on Personalized PageRank. Personalized PageRank is an extension of the famous *PageRank* algorithm [14], both of which are based on a random surfer model.

To understand Personalized PageRank, we first review the original PageRank briefly. A random surfer starts at any node on the graph. At each step, with a probability of $1 - \alpha$ the surfer moves to a neighboring node randomly, and with a probability of α she gets bored and teleports to a random node on the graph. This process is repeated until the walk converges to a steady-state. The stationary probability of the surfer at each node is taken as the score of the node. However, this form of score is purely based on the static link structure, indicating the overall popularity of each node on the graph, without tailoring to a specific query node.

In contrast, Personalized PageRank enables query-sensitive ranking, in the sense that we can specify a query node to obtain a "personalized" ranking accordingly. It is based on the same random surfer model of the original PageRank, except when the surfer teleports, she always prefers the query node q. Specifically, at each step, with probability α the surfer teleports to q instead of a random node, thus visiting the neighborhood of q more frequently. Thus, the stationary distribution, called a Personalized PageRank Vector (PPV), is biased towards q and its neighborhood, which can be interpreted as a popularity or relevance metric specific to q. We denote the PPV *w.r.t.* a query node q by $\mathbf{r}_q$, and $\mathbf{r}_q(p)$ refers to the entry corresponding to node p in $\mathbf{r}_q$, *i.e.*, p's score *w.r.t.* q.

More generally, a query q can comprise multiple nodes on the graph, such that in the teleportation the surfer can jump to any node in q. Fortunately, the computation for a multi-node query is no more difficult than for a single-node query due to the *Linearity*

*This material is based upon work partially supported by NSF Grant IIS 1018723, the Advanced Digital Sciences Center of the University of Illinois at Urbana-Champaign, and the Agency for Science, Technology and Research of Singapore. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the funding agencies.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 39th International Conference on Very Large Data Bases, August 26th - 30th 2013, Riva del Garda, Trento, Italy.
Proceedings of the VLDB Endowment, Vol. 6, No. 6
Copyright 2013 VLDB Endowment 2150-8097/13/04... $ 10.00.

Theorem [12, 8, 6], as the PPV *w.r.t.* a multi-node query is a simple linear combination of the individual PPV *w.r.t.* each node in the query. Hence, our discussion only covers single-node queries.

Challenges in efficiency. Unfortunately, computing an *exact* PPV is, in general, infeasible even on a moderately large graph due to the prohibitive time or space cost [8, 12]. To make exact computation manageable, early works [11, 12] restrict personalization (*i.e.*, the query) to only some selected nodes. While such partial personalization is in some cases acceptable, most applications demand full personalization, which supports any arbitrary node as queries. Thus, some recent efforts [12, 8, 5, 6, 7] propose full personalization methods for *approximate* PPVs. They trade accuracy for faster query processing by reducing the computation in consideration online, as well as resorting to partial precomputation offline, which we will further elaborate in Sect. 2. However, in these schemes, once the offline precomputation is completed at a predetermined approximation level, the trade-off between efficiency and accuracy cannot be easily controlled dynamically.

Our proposal. In this paper, we present FastPPV, an approximate algorithm for computing fully personalized PPV. To highlight, it features *incremental* and *accuracy-aware* query processing, which means we can control the trade-off between efficiency and accuracy online. The key insight to achieve such a control hinges on the novel concept of *scheduled approximation*—we "organize" the random walk paths to be considered in some meaningful layers, such that the approximation can be incremented layer by layer, and more layers render better accuracy.

In particular, we develop this scheduled approximation upon an existing concept called *inverse P-distance* [12]. As shown previously [12], a node p's score in the PPV *w.r.t.* a query node q equals to the inverse P-distance from q to p, which is the *reachability* from q to p through all possible tours (*i.e.*, paths):

$$\mathbf{r}_q(p) \equiv \sum_{t \in \{q \rightsquigarrow p\}} R(t), \tag{1}$$

where a tour $t \in \{q \rightsquigarrow p\}$ is a sequence of edges from q to p that may contain cycles. $R(t)$, the *reachability* of t, is the probability of reaching p from q through tour t in a random walk. For a tour t of the form $v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_{\mathcal{L}(t)}$ with length $\mathcal{L}(t)$,

$$R(t) \triangleq (1-\alpha)^{\mathcal{L}(t)} \cdot \alpha \cdot \prod_{i=0}^{\mathcal{L}(t)-1} \frac{1}{|\text{Out}(v_i)|}, \tag{2}$$

where $\alpha \in (0,1)$ is the teleporting probability in the random surfer model, and $|\text{Out}(v_i)|$ is the out-degree of v_i.

We note that inverse P-distance was previously explored [12] to decompose the computation of a PPV. Specifically, they use inverse P-distance to compute some PPV components for a restricted set of nodes, which are then assembled to obtain the final PPVs *w.r.t.* those restricted nodes. Thus, their goal is to compute exact PPVs, but only for a fixed subset of nodes, lacking full personalization. While identified as their future work [12], devising an approximate algorithm for full personalization, solely based on their original use of inverse P-distance without exploiting other properties, appears implausible. In this paper, we solve this problem by making a new observation on inverse P-distance—the tours in Eq. 1 are not equally important in contributing to the computation. This observation prompts us to investigate the novel principle of scheduled approximation by *partitioning and prioritizing* tours, which leverages inverse P-distance in a distinct way, as we discuss next.

Principle (Sect. 3). We *partition* the set T of all tours involved in inverse P-distance (Eq. 1) into disjoint subsets $T = T^0 \cup \cdots \cup T^\eta$ according to their contribution, and *prioritize* them to tackle a more "important" partition T^i earlier. While simple, this partition-and-prioritize principle has remained unexplored for PPV computation to this date, and possesses two ideal properties:

- *Incremental*: FastPPV processes tours partition by partition, starting from T^0. In iteration-i, it covers tours in T^i to compute an increment $\hat{\mathbf{r}}_q^i$, adding to the overall estimate $\hat{\mathbf{r}}_q = \hat{\mathbf{r}}_q^0 + \cdots + \hat{\mathbf{r}}_q^i$. As it covers more partitions, the error (in terms of L1 norm) monotonically decreases, and $\hat{\mathbf{r}}_q$ asymptotically approaches $\mathbf{r}_q$.
- *Accuracy-aware*: In each iteration, we show that the current L1 error is determinable using only the current estimate, even without knowing the exact PPV. Thus, this error can be utilized as a stopping condition to control the trade-off between efficiency and accuracy at query time.

Realization (Sect. 4). To realize the basic principle of partitioning tours, we propose a novel notion, *hub length*. Given some *hub nodes* H selected from V, we measure the hub length of a tour t, or $\mathcal{L}_h(t)$, as the number of hubs traversed by t. Then, we partition each T^i to contain tours of $\mathcal{L}_h(t) = i$. With a carefully selected $|H|$, such partitioning has two desirable properties to enable efficient computation:

- *Discriminating*: By choosing nodes of high out-degrees as hubs, from Eq. 2, the fewer hubs t contains, the larger t's reachability becomes in general. Thus, tours in an earlier partition T^i (with i hubs) tend to contribute more than those in a later partition T^{k+m} (with more than k hubs), allowing us to efficiently focus on the first few partitions that are more important for an accurate estimation. We formally prove an error bound that decreases exponentially as more partitions are covered.
- *Sharing*: By choosing "popular" nodes on the graph as hubs, *i.e.*, hubs are more reachable from many tours, different tours will share the same hubs. This sharing enables the reuse of common sub-structures to speed up computation. First, as the tour segments between hub nodes are shared, we can thus precompute and index their reachabilities, which we call *prime PPVs*, as *building blocks* to assemble an arbitrary tour at query time. Second, as we build partitions by hub length, it can be shown that tours in partition T^i simply extend those in T^{i-1} as prefixes, and thus successive iterations can reuse these prefixes.

Overall framework (Sect. 5). Upon the hub length-based realization, we devise an overall framework for FastPPV:

- *Offline precomputation*: We identify a desirable set of nodes as hubs, and precompute their prime PPVs as building blocks for online processing.
- *Online query processing*: We start from the tours of hub length 0 in T^0, and further process tours of increasing hub length in an iterative manner. Within each iteration, precomputed building blocks can be reused; across iterations, prefixes can be shared.

Empirical evaluation (Sect. 6). Finally, we conduct extensive experiments on two real-world datasets. We compare FastPPV with two competitive baselines [7, 8], and find out that FastPPV significantly outperforms them in both online and offline phases. More importantly, we are able to demonstrate the scalability of FastPPV on growing graphs. In particular, FastPPV can achieve a near-constant time query processing irrespective of graph size, through only a linear increase in the offline precomputation costs.

2. RELATED WORK

While Personalized PageRank [14, 12] enables a personalized or query dependent view of PageRank, its computation can be prohibitively expensive in time or space, for not only online but also

offline scenarios. Even on a moderately large graph, it is infeasible to compute PPVs online using the naïve iterative method. Alternatively, even with the Linearity Theorem [12], naïve precomputation of the exact PPV *w.r.t.* every node on the graph (*i.e.*, full personalization) is time consuming and requires at least $\Omega(|V|^2)$ bits to store, which is quadratic in $|V|$ or the number of nodes on the graph. It can be shown that the quadratic space complexity holds no matter how clever the compression scheme is [8].

Thus, designing efficient algorithms for personalized PageRank has become an important research area. While earlier work pursues exact computation by supporting partial personalization (*i.e.*, only a subset of nodes can be used in a query), our work aligns with more recent developments that aim at full personalization (*i.e.*, any node can be used in a query) at the cost of accuracy.

Exact, partial personalization. Haveliwala *et al.* first proposed topic-sensitive PageRank [11], which only precomputes 16 PPVs—each corresponds to a top level category in the Open Directory Project[1]. With the Linearity Theorem [12], finer-grained personalization can be supported, *e.g.*, in hub decomposition [12], intelligent surfer [17] and ObjectRank [4]. Despite this, full personalization is still infeasible on large graphs.

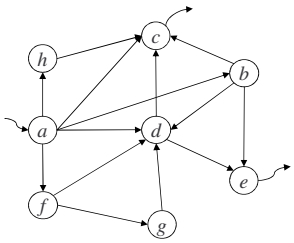
Approximate, full personalization. To achieve full personalization, most efforts resort to *approximate* computation instead. Intuitively and informally, the PPV *w.r.t.* a query q is a measure over random-walk paths starting from q. Thus, most of the existing approximation approaches can be perceived as a reduction in the total number of paths in their computation. First, only *hub-pivoted paths* that pass through some important "hub" nodes are considered, *e.g.*, Web Skeleton [12]. Second, only *sampled paths* using a Monte Carlo simulation are considered, *e.g.*, [8, 2]. Third, only *neighborhood paths* that are within some "radius" around q are considered, *e.g.*, Bookmark Coloring [5] and the HubRank family [6, 7, 16].

In addition, some techniques leverage additional properties, such as the block structure of the web [13], and the assumption of the power law distribution in PPV [3]. A few top-K methods have also been explored [10, 9], which often relies on bounds to identify the top K nodes without an actual estimate on node scores.

Connections to our work. Inspired by inverse P-distance [12] and the ideas of hubs and subgraph [5, 7], we introduce a novel principle of scheduled approximation by partitioning and prioritizing computation. This principle leads to our *incremental* query processing by iteratively enhancing the estimated PPV, which is also *accuracy-aware*—we can measure some form of error in our estimation after each iteration at query time. As we shall see in Sect. 3, this enables a dynamic trade-off between accuracy and efficiency at query time, which is lacking in previous works.

Specifically, as Sect. 1 explained, we make a novel observation on inverse P-distance [12] that tours are not equally important for they contribute to PPV computation differently. The importance of a tour can be quantified by *hub length*, which means our hubs serve a different purpose from previous works [5, 7]—to partition and prioritize computation, rather than to directly use their precomputed PPVs. As such, our offline precomputation is much cheaper, since we do not need to compute the PPVs for the hubs over the entire graph. We also use a subgraph at query time, but we are not computing the PPV on the subgraph directly as in previous works [5, 7]; instead, the subgraph is primarily used as a gateway to extend tours for efficient incremental enhancement.

[1]http://www.dmoz.org/



(a) Graph G

Tours (a to c)	Reachability
t_1: $a \to c$	$R(t_1) = 0.0255$
t_2: $a \to h \to c$	$R(t_2) = 0.0216$
t_3: $a \to d \to c$	$R(t_3) = 0.0108$
t_4: $a \to b \to c$	$R(t_4) = 0.0072$
t_5: $a \to f \to d \to c$	$R(t_5) = 0.0046$
t_6: $a \to b \to d \to c$	$R(t_6) = 0.0046$
t_7: $a \to f \to g \to d \to c$	$R(t_7) = 0.0017$

$\mathbf{r}_a(c)$

(b) $\mathbf{r}_a(c)$ computation (with teleporting prob. $\alpha = 0.15$)

Figure 1: PPV computation example.

3. PRINCIPLE: SCHEDULED APPROXIMATION

In this section, we propose the general principle of a "scheduled" PPV approximation method, which enables *incremental* and *accuracy-aware* query processing.

Running example. As our motivating example, we introduce a toy graph $G = (V, E)$ in Fig. 1(a), where $V = \{a, b, \ldots, h\}$ and $E = \{(a, b), (a, d), \ldots\}$. To simplify discussion, the example graph is unweighted and contains no cycles, although our framework works for a general graph with cycles.

Suppose the query node is a. By Eq. 1, $\mathbf{r}_a$ the PPV *w.r.t.* a captures the *reachability* from a to each node in G. Consider the personalized PR score $\mathbf{r}_a(c)$ for a specific node c (*i.e.*, the reachability from a to c), which can be computed by summing up the reachability of 7 tours, as illustrated in Fig. 1(b).

On a large graph, computing the reachability for all tours between each pair of nodes would cause serious efficiency issues. Fortunately, we have observed two facts that motivate an efficient PPV computation approach.

- *Some tours are more important than others in PPV computation*. A tour with higher reachability (*e.g.*, t_1) will rank its destination node (*e.g.*, c) highly by contributing more to the computation of the final score (*e.g.*, $\mathbf{r}_a(c)$).
- *Covering more tours in the computation would improve the accuracy*. For instance, if we handle more tours from t_1 to t_7, the cumulative reachability would be closer to the exact $\mathbf{r}_a(c)$.

The above two observations lead to the key insights of a scheduled approximation approach with two components:

- *Partitioning tours.* Instead of treating all tours equally, we first partition them into different *tour sets* according to their importance *w.r.t.* the query node. A partition of a full set of tours T is a set of disjoint subsets $T^0, \ldots, T^\eta$, where $T = T^0 \cup \ldots \cup T^\eta$ and $T^i \cap T^j = \emptyset, \forall i \neq j$.
- *Prioritizing computations.* Given a partition of tours $T = T^0 \cup \ldots \cup T^\eta$, we exploit the varying contribution of different tour sets, and schedule them for a prioritized PPV approximation—the most important set T^0 is traversed first for a fast estimate $\hat{\mathbf{r}}_q^{T^0}$ (*i.e.*, the aggregated reachability of tours in T^0 only), while less important ones are handled later to improve the accuracy incrementally. Note that we use $\hat{\mathbf{r}}_q$ to denote an estimated PPV to distinguish it from the exact one $\mathbf{r}_q$.

To be concrete, consider the graph G in Fig. 1(a). Suppose a is the query node and, for the purpose of illustration, we magically have the reachability of each tour at hand. Then, as Fig. 2 shows, we can partition the tours starting from a, $T = \{t_1, \ldots, t_{20}\}$, into some (say three) disjoint tour sets with decreasing importance by their reachability range: T^0, T^1, T^2. Note that in real scenarios,

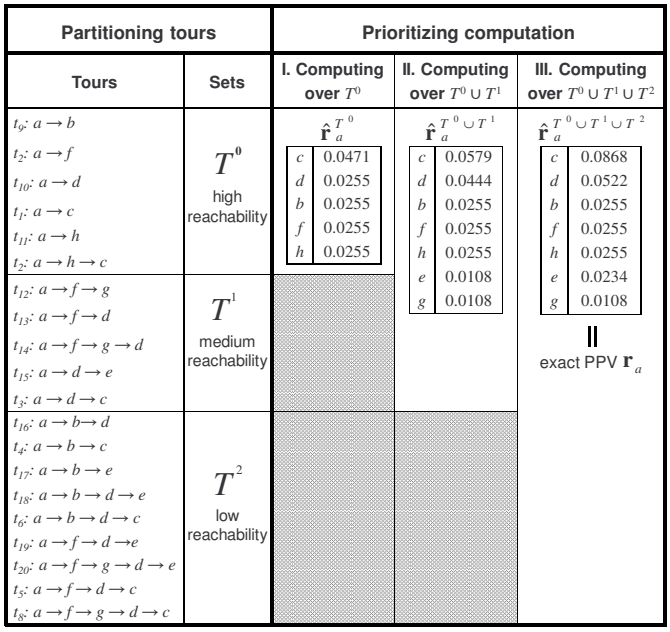


Figure 2: Scheduled approximation example.

we do not have the reachability of each tour beforehand; we will discuss a practical partitioning strategy in Sect. 4.

Now, to prioritize computation, we initially consider the most important set T^0 only for an estimated PPV $\hat{\mathbf{r}}_a^{T^0}$. According to $\hat{\mathbf{r}}_a^{T^0}$, the most relevant nodes to a (*i.e.*, c) have already been identified correctly. Subsequently, when T^1 is added, we obtain an enhanced estimate $\hat{\mathbf{r}}_a^{T^0 \cup T^1}$, which ranks the top five nodes c, d, b, f, h perfectly. Finally, when the tours in T^2 are also included, all the tours starting from a are covered, achieving the exact PPV $\mathbf{r}_a$.

This example illustrates a well-scheduled PPV computation process: T^0 is the top consideration since through T^0 most of the nodes in G, in particular the important ones, would be ranked; T^1 and T^2 are successively included to gradually improve our estimation. Towards the concept of scheduled computation, we propose an incremental query processing, which computes PPVs in a progressive manner where more time will render higher accuracy.

Incremental query processing. We estimate the PPV through multiple iterations, with each iteration handling an additional tour set, enhancing the overall approximation iteration by iteration.

More formally, given a partition of tours $T = T^0 \cup \ldots \cup T^\eta$ with decreasing importance, in iteration-i, a *PPV increment* $\hat{\mathbf{r}}_q^{T^i}$ is computed over the i-th important tour set T^i. For brevity, we also denote the increment by $\hat{\mathbf{r}}_q^i$. The overall approximation is the summation of all PPV increments from all iteration so far. That is, after iteration-k, we obtain an approximate PPV $\hat{\mathbf{r}}_q^{(k)}$:

$$\hat{\mathbf{r}}_q^{(k)} = \hat{\mathbf{r}}_q^{T^0 \cup T^1 \cup \ldots \cup T^k} = \sum_{i=0}^{k} \hat{\mathbf{r}}_q^i \tag{3}$$

Note that in $\hat{\mathbf{r}}_q^{(k)}$, the superscript (k) is enclosed in parentheses to mean that it is cumulative from T^0 to T^k, while the superscript in each PPV increment $\hat{\mathbf{r}}_q^i$ has no parentheses.

Such an incremental process enables flexible trade-off of efficiency and accuracy. As we process more iterations, the accuracy of approximation is gradually enhanced and if all tour sets are processed, an exact PPV is obtained. The reason is quite obvious. For a disjoint partition $T = T^0 \cup \ldots \cup T^\eta$, in iteration-$i$, tours in T^i are included in the computation. Thus more iterations will tackle more tour sets, and after iteration-η, each tour in T is covered for exactly once. This property is formalized by the following theorem.

Theorem 1. Given a query node q, let T be all tours starting from q and $T^0, \ldots, T^\eta$ be a partition of T. The estimated PPV score of any node $p \in V$ monotonically enhances with more iterations and eventually equals the exact one after iteration-η:

$$\hat{\mathbf{r}}_q^{(0)}(p) \leq \hat{\mathbf{r}}_q^{(1)}(p) \leq \ldots \leq \hat{\mathbf{r}}_q^{(\eta)}(p) = \mathbf{r}_q(p) \tag{4}$$

Generally, the graph can be cyclic, which contains a (countably) infinite number of tours. Thus, the partitioning might result in an infinite number of tour sets (*i.e.*, $\eta = \infty$) such that we need infinite iterations to achieve the exact PPV. However, we can expect an approximation which is arbitrarily accurate with sufficient iterations. In Sect. 4, given our specific partitioning and prioritizing methods, we would derive an error bound that is consistent with this expectation.

Accuracy-aware approximation. Due to the nature of our incremental processing, after each iteration, we can easily compute the L1 error of our estimation so far, even without knowing the exact PPV $\mathbf{r}_q$. The L1 error after iteration-k is defined as follows:

$$\varphi^{(k)} \triangleq \left\| \mathbf{r}_q - \hat{\mathbf{r}}_q^{(k)} \right\|_1 = \sum_{p \in V} \left| \mathbf{r}_q(p) - \hat{\mathbf{r}}_q^{(k)}(p) \right| \tag{5}$$

From Theorem 1, we know $\mathbf{r}_q(p) \geq \hat{\mathbf{r}}_q^{(k)}(p), \forall p, q \in V, \forall k \leq \eta$. Together with the fact that $\sum_{p \in V} \mathbf{r}_q(p) = 1$ (since $\mathbf{r}_q$ is a probability distribution over V), Eq. 5 can be re-expressed:

$$\varphi^{(k)} = \sum_{p \in V} \mathbf{r}_q(p) - \sum_{p \in V} \hat{\mathbf{r}}_q^{(k)}(p) = 1 - \sum_{p \in V} \hat{\mathbf{r}}_q^{(k)}(p) \tag{6}$$

The above equation provides a simple way to calculate $\varphi^{(k)}$ as the one's complement of the L1 norm of the current PPV estimate, without the knowledge of the final exact PPV. Thus, during online query processing, we can measure the L1 error after each iteration to enable a user-controllable trade-off between accuracy and time, *e.g.*, by specifying an accuracy requirement in terms of the L1 error, or a time limit for query processing.

Further, if we prioritize the tour sets appropriately, we can ensure that earlier iterations would bring in more improvement than later ones. Ideally, we should order the tour sets by their importance, or equivalently the sum of reachability of the constituent tours, *i.e.*, $\sum_{t \in T_0} R(t) \geq \ldots \geq \sum_{t \in T_\eta} R(t)$, which means that $\sum_{p \in V} \hat{\mathbf{r}}_q^0(p) \geq \ldots \geq \sum_{p \in V} \hat{\mathbf{r}}_q^\eta(p)$. By Eq. 6 and 3, this order will result in the largest reduction in L1 error after iteration-0, followed by iteration-1 and 2, and so on. Consequently, we can stop at an early iteration, yet still get the "most significant portion" out of the exact PPV. Formally, in Sect. 4, based on our actual partitioning and prioritizing strategy, we will prove an error bound that decreases exponentially as more iterations are processed.

However, while the principle is straightforward, the realization is challenging in two aspects:

- Challenge 1: *how can we partition and prioritize the tours?* In other words, how can we measure the importance of each tour? Naturally, we do not know the reachability of each tour beforehand—PPV computation is our ultimate goal, and thus it is impractical to partition the tours according to their reachability as we did in the example. We need a simple and unified metric, which can be efficiently applied to measure the importance of tours and is universally effective for different queries.

- Challenge 2: *how can we efficiently compute each PPV increment?* Computing each $\hat{\mathbf{r}}_q^i$ from scratch is not ideal since it is expensive to naïvely sum up the reachability of all tours involved. In our previous example in Fig. 2, we observe large overlaps between tours in different sets. *E.g.*, t_{12}, t_{13}, t_{14} in T^1 share an edge $a \to f$, which is a tour t_2 in T^0. Thus, we can take advantage of these overlaps to efficiently calculate each $\hat{\mathbf{r}}_q^i(p)$.

4. REALIZATION: FastPPV

As the next step, we tackle the two challenges in realizing the basic principle in Sect. 3. We seek an effective and simple partitioning-and-prioritizing metric, and an efficient algorithm for computing PPV increments.

To motivate both goals, let us take a deeper examination of our running example in Fig. 1. Observe that some nodes, like d, have two desirable properties for characterizing the importance of tours and enabling efficient PPV computations, respectively.

First, *Discriminating.* With many out-neighbors, d significantly decays the reachability of those tours passing through it, *i.e.*, it has a high "decaying power" due to Eq. 2. *E.g.*, for two resembling tours (with only one different node), $t_2 : a \to h \to c$ and $t_3 : a \to d \to c$, the reachability $R(t_3)$ is only $1/2$ of $R(t_2)$ due to the high decaying power of node d on t_3.

Second, *Sharing.* As many tours pass through d, the segments (a sequence of edges) around d may be shared in different tours. *E.g.*, the segment $f \to g \to d$ is shared by three tours starting from a as shown in Fig. 2. We say that d is "popular."

We refer to nodes with such properties as *hub nodes*, because topologically they look like *hubs* in a network, at the center of different connections. While the notion of hubs has been explored previously [12, 6, 7], we stress that our hubs serve dual unique purposes—as a crucial response to the dual challenges:

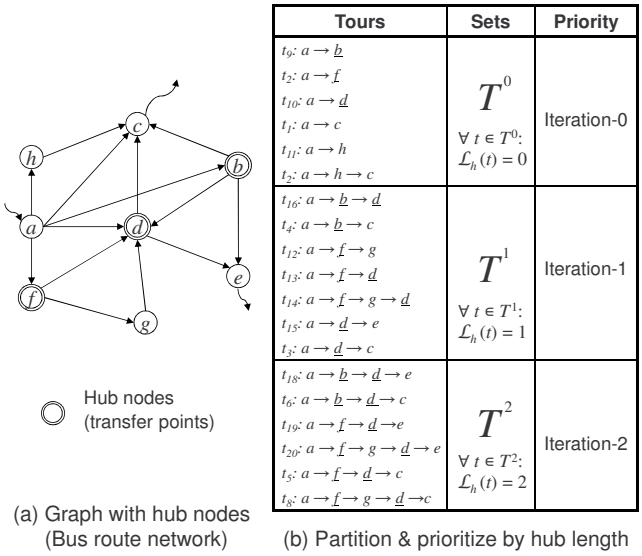
- *Tour set partitioning:* Hub nodes have high decaying power to discriminate tour importance and thus is a good criterion to partition tours (Challenge 1).
- *PPV increment computation:* Segments around hub nodes are shared by different tours and thus can be precomputed and reused to enable efficient computation (Challenge 2).

Next, we will discuss a hub selection strategy based on the above two properties. Afterwards, we will present an effective tour partition scheme (leveraging the discriminating property of hubs) in Sect. 4.1, and an efficient PPV computation algorithm (leveraging the sharing property of hubs) in Sect. 4.2.

Hub selection. To begin with, we quantify each criterion of a good hub—decaying power to discriminate tour importance, and simultaneously, popularity to share computation.

First, the *decaying power* of a node can be interpreted as its "utility" in discriminating tours, *i.e.*, how effective a node can discriminate tours passing through it. As shown in Eq. 2, the reachability of a tour decreases to a factor of $\frac{1}{|\text{Out}(v)|}$ when passing through a node v. That is, the high out-degree of v has a power to significantly decay the importance of a tour. Thus, we choose the out-degree of a node v to measure its decaying power, *i.e.*, its utility $U(v)$.

Second, the *overall popularity* of a node on the graph can be quantified by the commonly used PageRank score, as applied in a number of previous works [14, 12, 6]. We note that simpler alternatives exist too, such as in-degree, which is much cheaper to compute than PageRank. However, PageRank is more effective in capturing the "global" popularity of a node based on the entire graph, while the in-degree of a node only reflects its "local" popularity attributed by its direct in-neighbors. In addition, the cost of PageRank computation is not a major concern, since we only need to run it once offline, and its computation time is actually dominated by the other precomputation steps in the offline stage (see Sect. 5). Thus, we choose the PageRank score of a node v to measure its popularity, *i.e.*, the probability of v being "shared" in the computation, denoted by $P(v)$.



Tours	Sets	Priority
t_9: $a \to \underline{b}$ t_2: $a \to \underline{f}$ t_{10}: $a \to \underline{d}$ t_1: $a \to c$ t_{11}: $a \to h$ t_2: $a \to h \to c$	T^0 $\forall\, t \in T^0$: $\mathcal{L}_h(t) = 0$	Iteration-0
t_{16}: $a \to \underline{b} \to \underline{d}$ t_4: $a \to \underline{b} \to c$ t_{12}: $a \to \underline{f} \to g$ t_{13}: $a \to \underline{f} \to \underline{d}$ t_{14}: $a \to \underline{f} \to g \to \underline{d}$ t_{15}: $a \to \underline{d} \to e$ t_3: $a \to \underline{d} \to c$	T^1 $\forall\, t \in T^1$: $\mathcal{L}_h(t) = 1$	Iteration-1
t_{18}: $a \to \underline{b} \to \underline{d} \to e$ t_6: $a \to \underline{b} \to \underline{d} \to c$ t_{19}: $a \to \underline{f} \to \underline{d} \to e$ t_{20}: $a \to \underline{f} \to g \to \underline{d} \to e$ t_5: $a \to \underline{f} \to \underline{d} \to c$ t_8: $a \to \underline{f} \to g \to \underline{d} \to c$	T^2 $\forall\, t \in T^2$: $\mathcal{L}_h(t) = 2$	Iteration-2

Figure 3: Hub length-based tour partitioning and prioritizing.

Given a node v's utility $U(v)$ and probability $P(v)$, we can naturally define its *expected utility* $EU(v)$—an effective hub selection metric that integrates the discriminating and sharing properties:

$$EU(v) = P(v) \cdot U(v) = \text{PageRank}(v) \cdot |\text{Out}(v)|. \qquad (7)$$

Subsequently, given a desired number of hubs $|H|$ as an input parameter, we choose $|H|$ nodes with the largest expected utility as hubs. The setting of $|H|$ also impacts the performance, which we will analyze in Sect. 5 and validate in Sect. 6.

4.1 Tour Partitioning and Prioritizing

As the first challenge of scheduled PPV approximation, we need a partition scheme which is effective in discriminating tour importance and can be efficiently applied on the fly.

To motivate, consider our example graph G, assuming $H = \{b, d, f\}$ is the hub set for G. For intuition, we make an analogy that G is a bus transportation network, in which each node is a city and each edge is a bus route connecting two cities. To facilitate long distance transportation, some particular cities (*i.e.*, hub nodes in our approach) where multiple bus lines pass through, marked in a double circle, are selected as *transfer points*, as Fig. 3(a) shows.

Now suppose a passenger is planning a trip from city a to c. Which route is most likely to be chosen? Apparently, taking a *direct bus route* (which does not pass through any transfer point) between a and c, *i.e.*, $a \to c$ or $a \to h \to c$ (here h is merely a "stop-over", not a transfer point) is most preferred, followed by making one transfer (*e.g.*, $a \to \underline{d} \to c$), and then two transfers (*e.g.*, $a \to \underline{f} \to \underline{d} \to c$). Note that, when writing a tour, we underscore a node to stress a transfer point (or hub node).

Intuitively, just as people dislike routes that need many transfers, it is less likely to follow the tours containing more hub nodes in random walks, *i.e.*, these tours are less important in our prioritized PPV computation. More formally, each hub node would substantially decay the reachability of a tour passing through it due to its

large out-degree. The more hubs a tour passing through, the less important the tour is.

Partitioning and prioritizing by hub length. We formally quantify the importance of a tour by the number of hubs it passes through, which we call *hub length*.

Definition 1 (Hub Length). Given a set of hub nodes H, for any tour t, the hub length of t, denoted by $\mathcal{L}_h(t)$, is the number of hub nodes in t, excluding the starting and ending nodes.

Given this hub length metric, partitioning tours is straightforward. Consider the example graph with $H = \{b, d, f\}$ in Fig. 3(a). The tours starting from a are partitioned into three sets: T^0, T^1 and T^2, with each containing tours of a distinct hub length—the hub length of every tour is 0 in T^0, 1 in T^1 and 2 in T^2, as shown in Fig. 3(b). These tour sets form a valid partition—they are pairwise disjoint and cover all tours starting from a. Furthermore, the importance of tours in different sets are decreasing from T^0 to T^2, which naturally shows a desired order for prioritized computation.

More generally, given a set of hub nodes, for a query node q, we can partition all the tours rooted at q into η sets $T = T^0 \cup T^1 \cup \ldots \cup T^\eta$ such that, for $i \in [0, \eta]$, $T^i = \{t \,|t$ starts at $q \wedge \mathcal{L}_h(t) = i\})$, where η is the maximal hub length of all tours in T. Given such a partition, for a prioritized incremental computation, to return better results earlier, as Sect. 3 explained, the sets with a shorter hub length are handled earlier, in the order T^0 to T^η. As Sect. 3 also explained, for a cyclic graph, η can be infinite. However, tours with large hub length contribute trivially and, thus, can be omitted for a good approximation.

Error bound. Based on our partitioning and prioritizing strategy, we further exploit the L1 error of our incremental approximation discussed in Sect. 3. In particular, we formally prove a bound for the L1 error in each iteration, which further implies two theoretically desirable properties.

Theorem 2. After iteration-k, the L1 error $\varphi^{(k)}$ as defined in Eq. 5 satisfies the following bound:

$$\varphi^{(k)} \leq (1-\alpha)^{k+2} \tag{8}$$

PROOF. First, by Eq. 6 and 3, we have $\varphi^{(k)} = 1 - \sum_p \hat{\mathbf{r}}_q^{(k)}(p) = 1 - \sum_{i=0}^{k} \sum_{t \in T^i} R(t)$ [$*$].

Second, by Def. 1, $\forall t \in T^k, \mathcal{L}_h(t) = k$, and $\forall t, \mathcal{L}_h(t) < \mathcal{L}(t)$ where $\mathcal{L}(t)$ is the natural length of t (*i.e.*, the number of edges in t). Thus, if $\mathcal{L}(t) \leq k+1$, then $\mathcal{L}_h(t) \leq k$, implying $\cup_{i=0}^{k} T^i \supseteq \cup_{i=0}^{k+1} S^i$ where $S^i \triangleq \{t : \mathcal{L}(t) = i\}$. Hence, $\sum_{i=0}^{k} \sum_{t \in T^i} R(t) \geq \sum_{i=0}^{k+1} \sum_{t \in S^i} R(t)$ [$**$].

Third, we claim that $\sum_{t \in S^i} R(t) = (1-\alpha)^i \alpha$ [$***$], and show it by induction. The base case $i = 0$ is clearly true. In the induction step, suppose it is true for $i = \ell$. All tours with length $\ell+1$ must be extended from a tour of length ℓ by one step. Consider a particular tour t' of length ℓ. The total reachability of all tours of length $\ell+1$ that are extended from t' is $R(t')(1-\alpha)$ based on Eq. 2. Hence, $\sum_{t \in S^{\ell+1}} R(t) = \sum_{t' \in S^\ell} R(t')(1-\alpha) = (1-\alpha)^\ell \alpha (1-\alpha) = (1-\alpha)^{\ell+1}\alpha$, which proves the claim.

Finally, combining these results [$*, **, ***$], we can derive that $\varphi^{(k)} = 1 - \sum_{i=0}^{k} \sum_{t \in T^i} R(t) \leq 1 - \sum_{i=0}^{k+1} \sum_{t \in S^i} R(t) = 1 - \sum_{i=0}^{k+1} (1-\alpha)^i \alpha$, which simplifies to $\varphi^{(k)} \leq (1-\alpha)^{k+2}$. ∎

This bound exhibits two desirable properties. *First*, since $1 - \alpha < 1$, $\lim_{k \to \infty} \varphi^{(k)} = 0$. *Second*, the rate of $\varphi^{(k)}$ approaching zero is exponential as k grows. In other words, an earlier iteration contributes exponentially more than a later one. Thus, we can stop early yet still obtain a good estimation.

More specifically, for a typical $\alpha = 0.15$ as suggested in [14], we have $\varphi^{(10)} \leq 0.143$, $\varphi^{(20)} \leq 0.0280$ and $\varphi^{(30)} \leq 0.00552$, which diminishes exponentially as k increases. We note that, as the proof of Theorem 2 builds upon, we bound the error of the overall reachability of $\cup_{i=0}^{k} T^i$ by that of $\cup_{i=0}^{k+1} S^i$. In practice, $\cup_{i=0}^{k} T^i$ contains many more tours than $\cup_{i=0}^{k+1} S^i$, which makes the error to converge even faster. Our experiments in Sect. 6 show that as few as three iterations yield a very accurate PPV.

4.2 Efficient PPV Increment Computation

We next tackle the second challenge—to efficiently compute the PPV increment $\hat{\mathbf{r}}_q^i$ in iteration-i, which aggregates the reachability of tours of hub length i. Towards its efficient realization, we analyze the structure of tours in aggregation and develop a tour assembly model– interestingly, since tours are built from segments, the aggregation of tours into PPV amount to the aggregation of PPVs of the constituent segments around hubs. Next, we propose the sharing and reusing of such "hub-segment PPVs" in aggregation and, the "overlapping" of aggregations in different rounds, to enable an efficient PPV-increment computation.

4.2.1 Structured Aggregation: Tour Assembly Model

To enable efficient computation of $\hat{\mathbf{r}}_q^i$, we develop a tour assembly model to aggregate the similar segments in different tours, so that the overall reachability can be aggregated by such structured components rather than each individual tour.

Recall the bus transportation analogy in Sect. 4.1 (Fig. 3(a)). Consider the possible itineraries with two transfers from a to c– $t_5 : a \to \underline{f} \to \underline{d} \to c$, $t_6 : a \to \underline{b} \to \underline{d} \to c$ and $t_7 : a \to \underline{f} \to g \to \underline{d} \to c$. We observe that all these itineraries can be constructed by three "direct bus routes": one from the source a and two from subsequent transfer points. *E.g.*, t_7 is built by $a \to \underline{f}$ (a direct route from a), and two direct routes $\underline{f} \to g \to \underline{d}$ and $\underline{d} \to c$ from transfer points f and d respectively.

Our $\hat{\mathbf{r}}_q^i$ computation shares the same insight with this analogy of building itineraries. For a query node q, by viewing each tour from q as an itinerary starting at q and going through hubs as making transfers, we can "assemble" the reachability of any tour by combining the reachability of a *direct segment* (*i.e.*, tours passing through no hubs) from q to its nearest hub node and then several direct segments from each hub on the tour. Specifically, let's examine the tours just mentioned (t_5, t_6, t_7). For each tour, we can calculate its reachability by assembling three direct segments as follows:

$$\begin{aligned} R(t_5) &= \frac{1}{\alpha^2} \cdot R(a \to \underline{f}) \cdot R(\underline{f} \to \underline{d}) \cdot R(\underline{d} \to c) \\ R(t_6) &= \frac{1}{\alpha^2} \cdot R(a \to \underline{b}) \cdot R(\underline{b} \to \underline{d}) \cdot R(\underline{d} \to c) \\ R(t_7) &= \frac{1}{\alpha^2} \cdot R(a \to \underline{f}) \cdot R(\underline{f} \to g \to \underline{d}) \cdot R(\underline{d} \to c) \end{aligned}$$

Note that by the reachability definition (Eq. 2), at the end of a segment, the random surfer would stop with a probability α. Thus, to continue the tour, we need to compensate a probability α at each "transfer," *i.e.*, the $\frac{1}{\alpha^2}$ term in our two-transfer example above.

With such an "assembly" of individual reachabilities, we now build a systematic understanding of assembling $\hat{\mathbf{r}}_q^i(p)$. As an example, we will consider the above tours from a to c, to assemble $\hat{\mathbf{r}}_a^2(c)$. The result can be derived, step by step, as Eq. 9 shows.

$$\begin{aligned} \hat{\mathbf{r}}_a^2(c) &\triangleq R(t_5) + R(t_6) + R(t_7) = \big(R(t_5) + R(t_7)\big) + R(t_6) \\ &=^1 \frac{1}{\alpha^2} \cdot \hat{\mathbf{r}}_a^0(f) \cdot \hat{\mathbf{r}}_f^0(d) \cdot \hat{\mathbf{r}}_d^0(c) + \frac{1}{\alpha^2} \cdot \hat{\mathbf{r}}_a^0(b) \cdot \hat{\mathbf{r}}_b^0(d) \cdot \hat{\mathbf{r}}_d^0(c) \\ &=^2 \frac{1}{\alpha^2} \cdot \sum_{h_2 \in \mathcal{H}'(h_1)} \sum_{h_1 \in \mathcal{H}'(a)} \hat{\mathbf{r}}_a^0(h_1) \cdot \hat{\mathbf{r}}_{h_1}^0(h_2) \cdot \hat{\mathbf{r}}_{h_2}^0(c) \end{aligned} \tag{9}$$

To begin with, as different tours may share the same hubs (recall the "sharing" property), we wonder if we can first aggregate such tours to factor out their common segments? Let's re-examine the tours t_5, t_6, and t_7. As Fig. 4 shows, t_5 and t_7 share the same hubs f, d. Thus, if we merge them at each hub node, we can aggregate the reachability of individual segments in different tours (*e.g.*, $R(f \rightsquigarrow d)$ in t_5, $R(f \rightsquigarrow g \rightsquigarrow d)$ in t_7) into an overall reachability between the ending nodes (*e.g.*, $\hat{\mathbf{r}}_f^0(d)$). We can transform t_6 similarly, since it is segmented by another set of hubs—all by itself. This assembling is illustrated in step-1 of Eq. 9.

More generally, as we observed, aggregating those tours with the same hubs is effectively aggregating direct segments between hubs. This will prove to be useful, since we have now abstracted the aggregation of tours in terms of the set of hubs they pass through (in this case, from a to c through $\{f, d\}$ or $\{b, d\}$).

Further, to aggregate these "hub-abstracted" tours, we wonder if they can be enumerated in a systematic order. From the result of step 1, we observe that even the tours segmented by different hubs can be generated by a same two-level expansion pattern: $a \rightsquigarrow h_1 \rightsquigarrow h_2 \rightsquigarrow c$ where h_1 is the first-level hub (*e.g.*, f and d) to be reached from a, and h_2 is any hub (*e.g.*, d) to be reached at the second level (from h_1). To emphasize this level-by-level property, we refer to the first-level hubs h_1 as the *neighboring hubs* of the starting node a, denoted $\mathcal{H}'(a)$, and similarly, the second-level hubs are referred as the neighboring hubs of h_1, *i.e.*, $\mathcal{H}'(h_1)$. Therefore, to aggregate every tour in the form $a \rightsquigarrow h_1 \rightsquigarrow h_2 \rightsquigarrow c$, we further merge the neighboring hubs $\mathcal{H}'(h_i)$ at each level i as step 2 of Eq. 9 shows.

Overall, with our tour assembly model, we can transform the aggregation of tours into the aggregation of intermediate segments between hubs at each level. Specifically, by merging the segments from a to the first-level hubs (*e.g.*, f and d), we gather the tours in the "neighborhood" of a (*i.e.*, tours in T^0) to form a *prime subgraph* which is rooted at a and bordered by h_1's, denoted $\mathcal{G}'(a)$. We call the aggregated reachability of the constituent tours in $\mathcal{G}'(a)$ the *prime PPV* of a, denoted $\hat{\mathbf{r}}_a^0$. Similarly, merging tour segments from h_1 to the second-level hubs, we form the prime subgraphs of h_1 (*e.g.*, $\mathcal{G}'(f)$ and $\mathcal{G}'(d)$).

Definition 2 (Prime Subgraph and Prime PPV). Given a graph G and a set of hub nodes H, for any node v,

- The *prime subgraph* $\mathcal{G}'(v)$ of v consists of all the nodes and edges in T^0, the tours starting at v with $\mathcal{L}_h(t) = 0$; and the neighboring hubs of v, $\mathcal{H}'(v)$, can also be referred as the *border hub nodes* of $\mathcal{G}'(v)$.
- The reachability from v to each node through all tours in $\mathcal{G}'(v)$ forms the *prime PPV* of v, *i.e.*, $\hat{\mathbf{r}}_v^0$.

In general, we can compute $\hat{\mathbf{r}}_q^i(p)$, the reachability between q and p over tour set T^i by assembling $\hat{\mathbf{r}}_q^0$ and the prime PPVs of up to i-th level hubs, as formalized in Theorem 3. The essence of this theorem boils down to the Chapman-Kolmogorov equation [15], which relates a joint probability distribution with the combination of a set of transition probabilities.

Theorem 3. Let q be the query node, H be a set of hub nodes in graph G. For any node p, the personalized PR score estimated over tours of hub length i can be constructed as:

$$\hat{\mathbf{r}}_q^i(p) = \frac{1}{\alpha^i} \cdot \sum_{h_i \in \mathcal{H}'(h_{i-1})} \cdots \sum_{h_1 \in \mathcal{H}'(q)} \hat{\mathbf{r}}_q^0(h_1) \cdots \hat{\mathbf{r}}_{h_{i-1}}^0(h_i) \cdot \hat{\mathbf{r}}_{h_i}^0(p) \quad (10)$$

4.2.2 Computing with Prefixes and Building Blocks

By the tour assembly model discussed in Sect. 4.2.1, we are able to assemble a PPV increment by structured building blocks: the prime PPVs of hub nodes. We will now exploit the common substructures between PPV increments (calculated in successive iterations) for efficient computation.

To motivate, we rewrite Eq. 9 to connect $\hat{\mathbf{r}}_a^2(c)$ with the preceding PPV increments. To better illustrate this connection, we represent the border hubs in terms of their hub length, *e.g.*, $h_1 \in \mathcal{H}'(q)$ is explicitly represented as $\mathcal{L}_h(a \rightsquigarrow h_1) = 1$. Then, we rearrange and isolate the terms related to h_1 as an inner summation, which can be substituted with $\hat{\mathbf{r}}_a^1(h_2)$ (by Theorem 3), as follows:

$$\begin{aligned}
&\hat{\mathbf{r}}_a^2(c) \\
=&\frac{1}{\alpha} \cdot \sum_{h_2 \in H, \mathcal{L}_h(a \rightsquigarrow h_2)=1} \left(\frac{1}{\alpha} \cdot \sum_{h_1 \in H, \mathcal{L}_h(a \rightsquigarrow h_1)=0} \hat{\mathbf{r}}_a^0(h_1) \cdot \hat{\mathbf{r}}_{h_1}^0(h_2) \right) \cdot \hat{\mathbf{r}}_{h_2}^0(c) \\
=&\frac{1}{\alpha} \cdot \sum_{h_2 \in H, \mathcal{L}_h(a \rightsquigarrow h_2)=1} \hat{\mathbf{r}}_a^1(h_2) \cdot \hat{\mathbf{r}}_{h_2}^0(c) \qquad (11)
\end{aligned}$$

Basically, this transformation facilitates an efficient computation of PPV increments. To compute PPV increment-2 $\hat{\mathbf{r}}_a^2(c)$, we do not need to assemble the prime PPVs $\hat{\mathbf{r}}_a^0(h_1)$, $\hat{\mathbf{r}}_{h_1}^0(h_2)$, $\hat{\mathbf{r}}_{h_2}^0(c)$ from scratch; rather, it can be directly built upon PPV increment-1 $\hat{\mathbf{r}}_a^1(h_2)$ and a specific PPV $\hat{\mathbf{r}}_{h_2}^0(c)$ involved in iteration-2. Likewise, $\hat{\mathbf{r}}_a^1(h_2)$ itself can be assembled by PPV increment-0 $\hat{\mathbf{r}}_a^0(h_1)$ and a specific PPV $\hat{\mathbf{r}}_{h_1}^0(h_2)$, as illustrated in the inner summation of the first line in Eq. 11.

Formally, in Theorem 4 we present a general result by recursively expanding Eq. 10 for each iteration i. For any $\hat{\mathbf{r}}_q^i$ ($i > 0$), we can reuse $\hat{\mathbf{r}}_q^{i-1}$ computed over T^{i-1}, directly assembling it with the prime PPVs of the i-th level hub nodes in T^i, *i.e.*, $\hat{\mathbf{r}}_{h_i}^0(p)$. The proof mirrors the above derivation of Eq. 11.

Theorem 4. Let q be the query node, H be a set of hub nodes in graph G. For any node p, the personalized PR score estimated over T^i can be computed as:

$$\hat{\mathbf{r}}_q^i(p) = \frac{1}{\alpha} \cdot \sum_{h_i \in H, \mathcal{L}_h(q \rightsquigarrow h_i)=i-1} \hat{\mathbf{r}}_q^{i-1}(h_i) \cdot \hat{\mathbf{r}}_{h_i}^0(p) \quad (12)$$

In summary, Theorem 4 exploits the shared substructures both within and across iterations, entailing two crucial aspects for speeding up computation:

- *Reusing Prefix Tours*: The PPV increment-i or $\hat{\mathbf{r}}_q^i(p)$, computed over tours in T^i, can be simply extended from its prefix $\hat{\mathbf{r}}_q^{i-1}(h_i)$, which is already computed in $\hat{\mathbf{r}}_q^{i-1}$, the PPV increment of the last iteration. Thus, the incremental PPV enhancement can be efficiently realized by recursively reusing the PPV increment in a earlier iteration to construct an enhanced estimation.
- *Precomputing Building Blocks*: The extension beyond the prefix is $\hat{\mathbf{r}}_{h_i}^0(p)$, the prime PPV of hub h_i, which is independent of the query q. Thus, to enable fast online computation, we can precompute these query-independent prime PPVs (*i.e.*, prime PPV of each hub) and use them as *building blocks* to construct any PPV increment on the fly.

5. OVERALL FRAMEWORK

To materialize the computation in Eq. 12, our overall framework consists of two phases: 1) *Offline precomputation* where we precompute the building blocks; 2) *Online query processing* where we reuse the building blocks and prefix tours to incrementally compute a gradually more accurate PPV for any query.

To develop the two phases, we first treat the graph as residing in the main memory, a typical assumption adopted in recent works such as [6, 7]. Next, to handle graphs that are too large for the main memory, we propose a disk-based implementation for FastPPV.

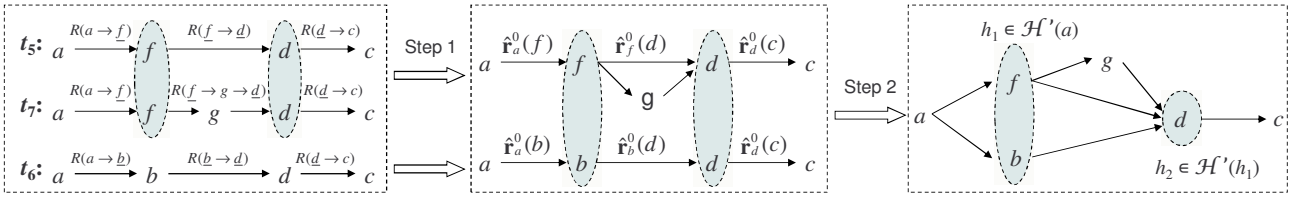


Figure 4: Tour assembly example, corresponding to the two steps in Eq. 9.

5.1 Offline Precomputation

In the offline phase, we need to compute the building blocks, *i.e.*, prime PPVs for a set of hub nodes H over G. These building blocks are then stored in an index, which will be used in online query processing.

Given a graph G and number of hubs $|H|$ as input, we first select hubs according to their expected utility (see Sect. 4). Next, for each $h \in H$, we compute its prime PPV $\hat{\mathbf{r}}_h^0$ using the standard iterative algorithm over its corresponding prime subgraph, which is feasible as prime subgraphs are many orders smaller than the entire graph. The prime subgraph can be identified using a depth-first search starting from the query node. During the search, we backtrack when we hit a hub node (which are the border hub nodes for this prime subgraph), or a "faraway" node whose reachability to the query node is smaller than a threshold ϵ (say 10^{-8}).

The above steps are summarized in Algorithm 1. In particular, the precomputed prime PPVs or building blocks are stored in a PPV index on disk, which can be loaded into the main memory as needed during online query processing.

Time complexity. As hubs essentially form the borders of prime subgraphs, we can informally view a graph as being divided into prime subgraphs at the hub nodes. Intuitively, more hubs result in smaller prime subgraphs. As each hub node blocks an entire search subtree during the depth-first search for the prime subgraph, the size of a prime subgraph decreases exponentially in $|H|$. Hence, it is reasonable to assume that on average a prime subgraph is smaller than $O(1/|H|)$ of the entire graph, *i.e.*, contains fewer than $O(|V|/|H|)$ nodes and $O(|E|/|H|)$ edges. Thus, computing a prime PPV over such a prime subgraph using the standard iterative algorithm costs less than $O\left(I(|V|+|E|)/|H|\right)$, where I is the number of iterations. Therefore, the total precomputation time for all hubs can be upper bounded by $O\left(|H| \cdot I(|V|+|E|)/|H|\right) = O\left(I(|V|+|E|)\right)$. This implies that our offline precomputation is scalable in the number of hubs, since the upper bound is independent of $|H|$. The ability to index a large number of hubs offline is crucial to speeding up online query processing as we will discuss in Sect. 5.2. Although our time complexity is under a quite simplifying assumption, the experiments in Sect. 6.3 do demonstrate that offline precomputation is scalable in the number of hubs.

Space complexity. Under the same assumption that on average a prime subgraph is smaller than $O(1/|H|)$ of the entire graph, the space cost of the PPV index can be likewise upper bounded by $O\left(|H| \cdot |V|/|H|\right) = O(|V|)$, which is also independent of $|H|$.

5.2 Online Query Processing

In the online phase, given a graph G, a precomputed PPV index Φ, a query node q and a stopping condition S, we incrementally compute an approximate PPV $\hat{\mathbf{r}}_q^{(\eta)}$ for a given query q according to Eq. 12, as sketched in Algorithm 2. The algorithm consists of two major steps: computing the initial iteration $i = 0$ (line 1–5) and subsequent iterations $i \geq 1$ (line 6–16).

In the initial iteration $i = 0$, we need to compute the prime PPV $\hat{\mathbf{r}}_q^0$ for the query node q, which is required in iteration $i = 1$ (see Eq. 12). If q happens to be a hub node, we can directly load $\hat{\mathbf{r}}_q^0$ from the precomputed index; otherwise we need to compute it on-the-fly.

Algorithm 1: OfflinePrecomputation

Input: a graph G; number of hub nodes $|H|$
Output: PPV index Φ

```
1  Φ ← ∅;
2  H ← Select |H| hubs on G;
3  foreach h ∈ H do
4  |  Compute prime PPV r̂_h^0 for h on G;
5  |  Store r̂_h^0 in PPV index Φ;
6  end
7  return Φ.
```

Algorithm 2: OnlineQueryProcessing

Input: a graph G; PPV index Φ over H; query node q; stopping condition S
Output: estimated PPV $\hat{\mathbf{r}}_q^{(\eta)}$

```
1  if q ∉ H then
2  |  Compute prime PPV r̂_q^0 for q on G;
3  else
4  |  Load r̂_q^0 from PPV index Φ;
5  end
6  r̂_q^(η) ← r̂_q^0; i ← 0; H_exp ← H'(q);
7  while the stopping condition S not met do
8  |  i ← i + 1; r̂_q^i ← 0; H_nextToExp ← ∅;
9  |  foreach h ∈ H_exp such that r̂_q^{i-1}(h) > δ do
10 |  |  Load r̂_h^0 from PPV index Φ;
11 |  |  r̂_q^i ← r̂_q^i + (1/α) r̂_q^{i-1}(h) r̂_h^0;
12 |  |  H_nextToExp ← H_nextToExp ∪ H'(h);
13 |  end
14 |  r̂_q^(η) ← r̂_q^(η) + r̂_q^i;
15 |  H_exp ← H_nextToExp;
16 end
17 return r̂_q^(η).
```

To compute subsequent iterations $i \geq 1$ for Eq. 12, we will reuse the prefixes—PPV increment $\hat{\mathbf{r}}_q^{i-1}$ from iteration $i-1$, as well as the building blocks—precomputed prime PPVs of some hub nodes $h \in H_{\text{exp}}$ (line 11). In particular, these hub nodes $h \in H_{\text{exp}}$ are the border hub nodes of the hubs used in iteration $i-1$ (line 12).

It is worth noting that we also need to specify a stopping condition S as an input. The choice of S is flexible depending on the desired trade-off between accuracy and efficiency—we can stop the incremental iterations when an accuracy requirement (in terms of L_1 error) is achieved, or a time limit for query processing is up, or the maximum number of iterations η is reached.

For a practical implementation, we impose a threshold δ (say 0.005) on the border hub nodes, such that we include them only if $\hat{\mathbf{r}}_q^{i-1}(h) > \delta$ (line 9). This threshold prevents least contributing hubs, improving efficiency with minimal impact on accuracy.

Time complexity. Suppose there is an average of $O(|\bar{H}|)$ border hub nodes in each prime PPV. Thus, in η iterations we need to handle $O(|\bar{H}|^\eta)$ hub nodes. Typically $|\bar{H}| \ll |H|$ and $\eta \leq 5$. For instance, in our experiments, even when $\eta = 1$, an average precision

of above 0.9 can already be achieved. Hence, this complexity is practically feasible. Additionally, if the query is not a hub node, an extra $O((|V| + |E|)/|H|)$ time is needed for computing its prime PPV, which decreases with a larger $|H|$.

5.3 Disk-based Implementation

Many real-world graphs are too large to reside entirely in the main memory. To handle these graphs, we describe a disk-based approach for online query processing. Disk-based offline precomputation can be implemented using similar ideas.

First, we observe that in online processing, we need the entire graph in the main memory such that we can identify the prime subgraph for the query node. However, after we have obtained the prime subgraph, we no longer require the entire graph—only the prime subgraph is needed, which is generally many orders of magnitude smaller than the entire graph.

Hence, given a query node, the key to the disk-based online query processing is to identify its prime subgraph from a disk-resident graph. The basic idea is to segment the graph into a number of clusters, such that we can at least fit each single cluster into the main memory. Subsequently, we can assemble the prime subgraph by searching in each cluster separately.

Specifically, to identify the prime subgraph for the query node, we first load the cluster that contains the query node into the memory, and start the depth-first search in this cluster until we reach a node that is outside this cluster. We call this event a *cluster fault*, at which point we will swap the required cluster into the main memory to continue the depth-first search. As frequent cluster faults significantly slow down query processing, we may prematurely terminate the search once reaching a threshold on the number of cluster faults. This can considerably speed up query processing with a minimal loss in accuracy. In our experiments, we set the threshold to the total number of clusters, which is generally robust.

Finally, to segment a graph into clusters, we adopt the technique in [18]. Specifically, a number of "anchor" nodes are chosen randomly, and every other node in the graph is assigned to its "nearest" anchor in terms of their personalized PageRank *w.r.t.* the anchor. It has been shown that personalized PageRank exhibits a good clustering quality [1]. Hence, we can obtain tight clusters even though the anchors are selected randomly, since every node in a cluster can become the anchor.

6. EXPERIMENTS

We empirically evaluated FastPPV on two real-world graphs. The experiments showed that it substantially outperforms previous state-of-the-art baselines in both the offline and online phases. We also investigated the performance of FastPPV on larger or disk-resident graphs, and concluded that it is scalable.

Datasets. We used two public real-world graphs below. In particular, the first graph is undirected, and the second one is directed.

- DBLP[2]: A *bibliographic network* of authors, papers and venues, with *undirected* edges representing the author-paper and paper-venue relationships. The graph contains 2 million nodes and 8.8 million edges.
- LiveJournal[3]: A *social network* where users can declare their friends. The friendship relationship is not necessarily reciprocal, and hence a *directed* edge from node i to j means that user i declares j as a friend. We sampled a graph with 1.2 million nodes and 4.8 million edges. Larger samples will also be used to study the scalability of FastPPV in Sect. 6.4.

[2]Available at http://www.informatik.uni-trier.de/~ley/db/

[3]Available at http://snap.stanford.edu/data/

Test queries. We randomly chose 1000 nodes from each graph, where every chosen node is a test query. We only focus on these single-node queries, since a multi-node query can be easily decomposed as multiple single-node queries using the Linearity Theorem [12, 8, 6]. For each experiment, we report the average performance over all test queries.

Baselines. We compare FastPPV with two baselines, representing two major lines of techniques. In particular, HubRankP [7] is the most competitive work among those that rely on reusing computation, while MonteCarlo is based on the popular fingerprint work [8] that relies on Monte Carlo simulation.

We implemented HubRankP [7] using their proposed benefit-based hub selection model to optimize online query time. To realize their benefit model, we assume a uniformly distributed query log, which is fair as our test queries are also sampled uniformly. Note that HubRankP builds upon the Bookmark-coloring algorithm [5] with a better hub selection policy. In addition, as Chakrabarti *et al.* [7] show, HubRankP is also superior to HubRankD [6] (an improvement over Jeh and Widom's hub decomposition method [12]). Hence, among these various works [7, 6, 5, 12], we only present the state-of-the-art HubRankP as the baseline.

We implemented a second baseline MonteCarlo using fingerprints [8], which simulates random walks on a graph. Specifically, a *fingerprint* is a sample random walk path, and the more samples we obtain, the more accurate the approximation is. Although it was originally meant to sample fingerprints for each query node offline, we can process queries online by on-the-fly sampling. To reduce the online workload, we first sample fingerprints for a set of hub nodes offline, which can be reused online. To increase the chance of hitting a hub node, we select nodes with largest global PageRank scores as hubs, a common strategy used previously [12, 5].

Parameters. For FastPPV, we must specify the following two parameters as part of the input:

- *Number of hubs* $|H|$, which influences online query processing speed. We will specifically study the performance under different $|H|$. However, as default, for other experiments we empirically set $|H| = 20$K for DBLP and $|H| = 120$K for LiveJournal unless otherwise stated, such that online query times are comparable on both datasets.
- *Number of iterations* η, which dynamically controls the trade-off between accuracy and query time in the online phase. By default we use $\eta = 2$ unless otherwise stated.

For both baselines, we also need to set the number of hubs $|H|$. In addition, each of them also has a parameter to control the accuracy which affects both the online and offline phases. In particular, HubRankP relies on a residual threshold ϵ_{push}, and MonteCarlo depends on the samples per query N. We will discuss the choice of these parameters in the corresponding experiments.

Moreover, for all the methods, we clip their PPV at 10^{-4}, *i.e.*, discarding nodes with scores less than 10^{-4} for offline storage. It can drastically reduce offline space cost with minimal impact on accuracy, as shown previously [6, 7]. Finally, we set $\alpha = 0.15$ (Eq. 2), which is a typical teleporting probability.

Accuracy metrics. Given a query, all the methods compute an approximate PPV. Thus, we need to evaluate their accuracy *w.r.t.* the exact PPV, in terms of ranking and score. Since users are usually more interested in higher ranked nodes, we focus on the top 10 nodes. Our accuracy objective is two-fold—we evaluate not only node rankings, but also node scores. In particular, we adopted four metrics from previous works [6, 7, 8], namely *Kendall's* τ and *precision* to measure the rankings, as well as *RAG* and *L1 error* to measure the scores. We refer readers to [6] for details.

For a consistent presentation, we report the complement of L1 error ($1 - $ L1 error) instead, which we call *L1 similarity*. Now, all the metrics indicates a better accuracy with a larger value.

Environment. We implement all methods in Java, and evaluate them on a Linux system with 2.67GHz CPU and 10GB RAM. The entire graph resides in the main memory except for the disk-based implementation in Sect. 6.4. In that case, the graph is disk-resident as we assume a reduced memory budget.

6.1 Comparison to Baselines

We compare the performance of FastPPV and our baselines. As all the methods compute approximate PPVs, there is a trade-off between accuracy and query time. To demonstrate the edge of FastPPV over the baselines, we configure the parameters to moderate their accuracy, where FastPPV *achieves an accuracy level similar to or better than the baselines*. Moderating their accuracy in this way enables us to fairly compare FastPPV with the baselines in terms of their online query time and offline aspects.

Four such accuracy-moderated configurations have been identified in Fig. 5. In each configuration, all the methods share a parameter $|H|$, in addition to their individual parameter mentioned earlier. The moderated accuracy can be verified in Fig. 6.

	Dataset	all:$\|H\|$	HubRankP:ϵ_{push}	MonteCarlo:N	FastPPV:η
I	DBLP	20K	0.11	120K	2
II	DBLP	30K	0.13	40K	1
III	LiveJournal	150K	0.20	200K	3
IV	LiveJournal	200K	0.29	10K	1

Figure 5: Four accuracy-moderated configurations (I–IV).

We first examine the online query time in Fig. 7(a). Across the four accuracy-moderated configurations, FastPPV is $2.0 \sim 7.2\times$ faster than HubRankP, and $2.4 \sim 5.2\times$ faster than MonteCarlo.

Second, we examine the offline phase in Fig. 7(b)–(c). As illustrated in Fig. 7(b), FastPPV requires less space than MonteCarlo, whereas it needs up to 30% more space than HubRankP in config. I. In contrast, for the offline precomputation time in Fig. 7(c), FastPPV is $4.3 \sim 11.0\times$ faster than HubRankP, and $2.9 \sim 14.3\times$ faster than MonteCarlo. Recall that online processing by FastPPV is also much faster under these configurations. As storage is increasingly cheaper, we consider the 30% more space required by FastPPV acceptable to trade for the much faster offline precomputation and online processing.

In summary, FastPPV is superior in both online and offline phases among the three methods.

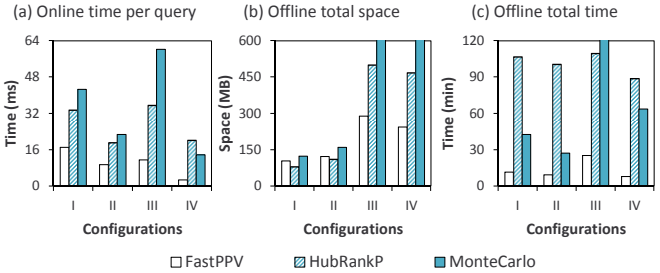


Figure 7: Accuracy-moderated comparisons with baselines.

6.2 Effects of Hub Selection Policy

We compare a few different hub selection policies for FastPPV. Apart from our policy of *expected utility* discussed in Sect. 4, we also select hubs by PageRank (popularity) or out-degree (utility) alone. Additionally, we also evaluate a random selection policy. However, its performance is substantially worse than the other policies, and hence we omit it here.

As different policies select different hub sets H, both offline precomputation for H and online processing using H are affected by different hub selections. To eliminate the effect of other parameters, we used the default number of hubs and iterations mentioned previously across all hub selection policies.

We first present the impact of different policies on online query processing in Fig. 8. While expected utility results in an accuracy level similar to or better than the others, it significantly speeds up online query processing—it is $1.2\times$ faster on DBLP and $2.4\times$ faster on LiveJournal than the second best policy. As DBLP is undirected, the three policies are fairly correlated with smaller differences than they are on the directed LiveJournal. Hence, the speed-up is more significant on LiveJournal.

Expected utility also results in cheaper offline precomputation for the selected hubs. In Fig. 9, while the space cost of expected utility is similar to other policies, precomputation is $1.3\times$ faster on DBLP and $1.7\times$ faster on LiveJournal than the second best policy. Likewise, the improvement is larger on the directed LiveJournal. Note that the precomputation time here includes the time to compute the prime PPVs for every hub, but excludes the time to select these hubs—the latter is negligible compared to the former.

These results clearly demonstrate that advantage of our proposed hub selection policy based on expected utility, which improves both the online and offline phases over the baseline policies.

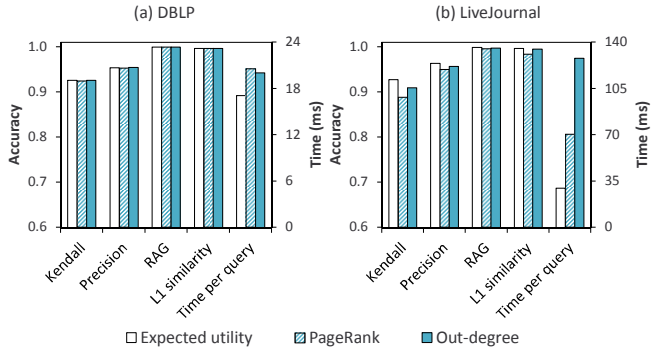


Figure 8: Effect of hub selection policies on online processing. Left axis: accuracy (Kendall, Prec, RAG, L1). Right axis: time.

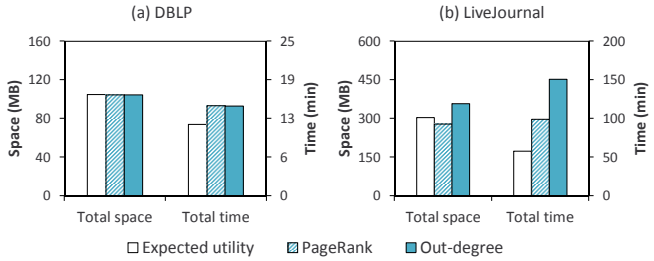


Figure 9: Effect of hub selection policies on costs of offline precomputation. Left axis: space cost. Right axis: time cost.

6.3 Effects of Parameters

We study how the number of hubs $|H|$ and iterations η affect the performance of FastPPV. In particular, we will fix one of the parameters with its default value, and vary the other.

6.3.1 Number of Hubs

We first illustrate the effect of varying the number of hubs $|H|$ on online query processing in Fig. 10. As expected, having more hub nodes drastically reduces the query time of FastPPV (see Sect. 5.2). Interestingly, even with a greatly reduced query time, all the accuracy metrics remain robust.

Config.	Kendall			Precision			RAG			L1 similarity		
	FastPPV	HubRankP	MonteCarlo	FastPPV	HubRankP	MonteCarlo	FastPPV	HubRankP	MonteCarlo	FastPPV	HubRankP	MonteCarlo
I	0.9255	0.9214	0.9211	0.9538	0.9517	0.9568	0.9993	0.9995	0.9999	0.9961	0.9845	0.9964
II	0.8894	0.8944	0.8904	0.9305	0.9354	0.9388	0.9988	0.9988	0.9997	0.9939	0.9780	0.9938
III	0.9281	0.9272	0.8864	0.9636	0.9628	0.9588	0.9989	0.9983	0.9998	0.9972	0.9774	0.9971
IV	0.8231	0.8237	0.7739	0.9178	0.9067	0.9192	0.9966	0.9886	0.9989	0.9908	0.9579	0.9880

Figure 6: FastPPV achieves an accuracy level similar to or better than the baselines under each accuracy-moderated configuration.

We further study the effect of $|H|$ on offline precomputation, as shown in Fig. 11. As $|H|$ grows, we observe that the total space cost increases sublinearly, whereas the total precomputation time actually decreases. Let us analyze such trends. As $|H|$ increases linearly, each prime subgraph becomes smaller exponentially (see Sect. 5.1). Hence, the total size of the prime subgraphs decreases, resulting in a decreasing total precomputation time. Likewise, we would also expect a decreasing total space cost, contrary to what we have observed. The reason is that we applied clipping on the prime PPVs, which is more effective on larger prime PPVs.

Hence, with a decreasing precomputation time and sublinearly increasing space cost, it is feasible to index more hubs offline, which also speeds up online query processing without compromising accuracy. Of course, if we index too many hubs (substantially more than what we are using now), the I/O overhead may eventually outweigh the benefit, since fetching the precomputed prime PPV of a hub node during online query processing requires one random access to the disk.

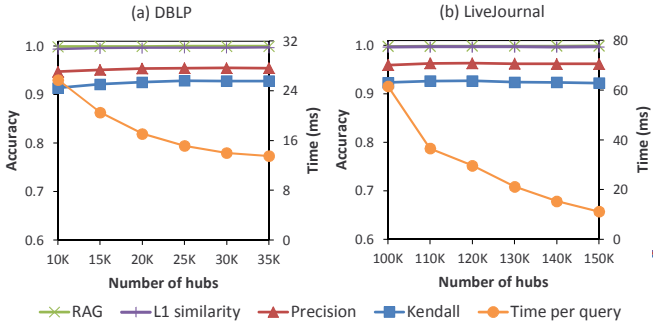


Figure 10: Effect of $|H|$ on online processing. Left axis: accuracy (RAG, L1, Prec, Kendall). Right axis: time.

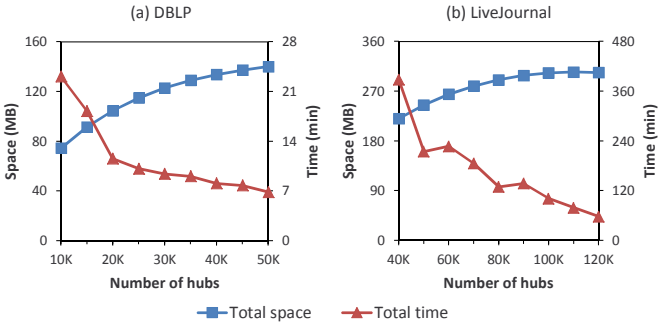


Figure 11: Effect of $|H|$ on costs of offline precomputation. Left axis: space cost. Right axis: time cost.

6.3.2 *Number of Iterations*

We explore FastPPV's *incremental* query processing by varying the number of iterations η.

As depicted in Fig. 12, allowing more iterations results in better accuracy but takes longer to process. This verifies that our approximation indeed becomes more accurate in an incremental manner. In particular, the improvement in accuracy is more significant in earlier iterations, which is consistent with Theorem 2. Thus, good accuracy can be achieved with only a few iterations. For instance, in Fig. 12 all the accuracy metrics are above 0.9 with $\eta = 2$ only.

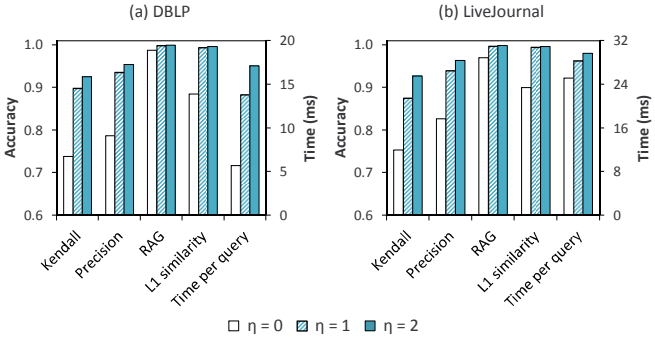


Figure 12: Incremental online processing by varying η. Left axis: accuracy (RAG, L1, Prec, Kendall). Right axis: time.

It is worth noting that η only affects online query processing, allowing us to dynamically control the trade-off between accuracy and query time without re-executing the offline phase. In contrast, many previous works including our baselines lack such flexibility. To adjust the trade-off, their offline phases may need a re-execution.

6.4 Scalability

We investigate the scalability of FastPPV in terms of two aspects. *First*, how does FastPPV scale on larger graphs in the online and offline phases? *Second*, how does the disk-based implementation fare on a disk-resident graph due to insufficient main memory?

6.4.1 *Scaling to Larger Graphs*

We first need to obtain graphs of varying sizes. On the one hand, each paper in DBLP has a timestamp. Thus, we take a snapshot of DBLP every four years from 1994 to 2010, as shown in Fig. 13(a). The snapshot graphs increase in size as time passes. On the other hand, we have no timestamp information in LiveJournal. Thus, we resort to sampling different numbers of edges from LiveJournal, as shown in Fig. 13(b). We order these sampled graphs in increasing size, and label them S1 to S5.

(a) Snapshots from DBLP

Snapshot year	# Nodes	# Edges
1994	0.32M	1.11M
1998	0.54M	2.00M
2002	0.88M	3.48M
2006	1.51M	6.40M
2010	2.00M	8.79M

(b) Samples from LiveJournal

Sample ID	# Nodes	# Edges
S1	0.31M	0.76M
S2	0.83M	2.67M
S3	1.22M	4.81M
S4	1.53M	7.01M
S5	1.77M	9.30M

Figure 13: Varying graph size for scalability study.

The key to scaling to larger graphs is to index more hubs offline. As shown in Fig. 14, even though the graph increases more than 5 folds on both datasets, by using a larger number of hubs $|H|$, we are able to achieve a *near constant online query time* without compromising accuracy. Hence, FastPPV can efficiently process queries online regardless of graph size, given sufficient number of hubs. In our study, we empirically determined the number of hubs required to achieve a constant query time over growing graphs. It is also

interesting to predict the requirement analytically, which warrants further investigation in a future work.

Next, we examine any additional cost involved in the offline phase in order to achieve a constant online query time. In Fig. 15, we plot the total space and time needed by offline precomputation against graph size (*i.e.*, the total number of nodes and edges). The plots clearly show a linear relationship between the total space (or time) and graph size. We deem such linear growths in the offline phase acceptable for maintaining a constant online query time.

(a) DBLP

Year	$\|H\|$	Kendall	Precision	RAG	L1 similarity	Time per query
1994	1K	0.9304	0.9520	0.9995	0.9966	**15.7 ms**
1998	3K	0.9245	0.9508	0.9993	0.9968	**16.1 ms**
2002	8K	0.9309	0.9556	0.9995	0.9965	**15.1 ms**
2006	15K	0.9286	0.9527	0.9993	0.9962	**15.7 ms**
2010	25K	0.9285	0.9545	0.9994	0.9963	**15.2 ms**

(b) LiveJournal

ID	$\|H\|$	Kendall	Precision	RAG	L1 similarity	Time per query
S1	14K	0.9274	0.9681	0.9984	0.9966	**28.5 ms**
S2	63K	0.9244	0.9637	0.9984	0.9970	**28.0 ms**
S3	120K	0.9269	0.9633	0.9985	0.9967	**29.7 ms**
S4	160K	0.9252	0.9645	0.9983	0.9965	**27.5 ms**
S5	200K	0.9210	0.9627	0.9986	0.9962	**29.9 ms**

Figure 14: Scaling FastPPV in online query processing.

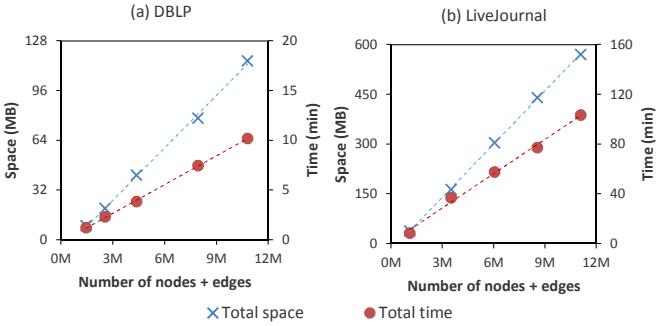


Figure 15: The costs of offline precomputation in order to scale FastPPV online. Left axis: space cost. Right axis: time cost.

6.4.2 *Disk-based Online Processing*

Assuming that our graphs do not fit into the main memory, we use our disk-based online processing, where a graph is segmented into a number of clusters to mimic the reduced memory budget (see Sect. 5.3). Recall that at any time, only one cluster needs to be in the memory. Hence, the size of the largest cluster is the minimum working set, which is much smaller than the entire graph.

As reported in Fig. 16, the disk-based implementation is scalable in the number of clusters. *First*, when we have more clusters, query time remains stable. Although cluster faults become more frequent with more clusters, the clusters also become smaller which are faster to swap into the main memory, resulting in similar query times. *Second*, as the largest cluster also shrinks with more clusters, the memory requirement decreases as well.

	(a) DBLP			(b) LiveJournal		
# Clusters	# Faults per query	Time per query	Memory need †	# Faults per query	Time per query	Memory need †
10	7.3	1434 ms	15.2%	6.8	747 ms	19.8%
15	10.8	1376 ms	10.3%	10.2	783 ms	15.1%
25	17.8	1370 ms	7.6%	16.8	862 ms	11.7%
35	24.7	1316 ms	5.4%	23.4	833 ms	6.4%
50	35.0	1270 ms	3.5%	33.3	831 ms	5.3%

† The size of the largest cluster, as a percentage of the entire graph.

Figure 16: Disk-based online query processing.

7. CONCLUSION AND FUTURE WORK

In this paper, we presented a scheduled approximation strategy to approximate PPVs. Specifically, we developed a hub length-based scheduling scheme for partitioning and prioritizing tours, as well as a structured aggregation model for assembling PPVs. As a result, our online processing is incremental and accuracy-aware, enabling a dynamic trade-off between efficiency and accuracy at query time. Empirically, FastPPV is not only superior to two state-of-the-art baselines, but also scalable.

As future work, we identify three major directions to explore. First, *automatic configuration*: for example, automatically determine the optimal number of hubs by correlating with various graph properties like density and diameter. Second, *tackling dynamic graphs*: as a graph can evolve over time, a simple idea to process graph updates is to only re-compute the affected prime PPVs, without touching the unaffected ones. Third, *generalizing to other graph algorithms*: it is promising to apply the same principle of partitioning and prioritizing tours to other random walk-based algorithms, such as the hitting and commute time measures.

8. REFERENCES

[1] R. Andersen, F. Chung, and K. Lang. Local graph partitioning using pagerank vectors. In *FOCS*, pages 475–486, 2006.

[2] B. Bahmani, K. Chakrabarti, and D. Xin. Fast personalized PageRank on MapReduce. In *SIGMOD*, pages 973–984, 2011.

[3] B. Bahmani, A. Chowdhury, and A. Goel. Fast incremental and personalized PageRank. *VLDB*, pages 173–184, 2010.

[4] A. Balmin, V. Hristidis, and Y. Papakonstantinou. ObjectRank: Authority-based keyword search in databases. In *VLDB*, pages 564–575, 2004.

[5] P. Berkhin. Bookmark-coloring algorithm for personalized pagerank computing. *Internet Mathematics*, 3(1):41–62, 2006.

[6] S. Chakrabarti. Dynamic personalized pagerank in entity-relation graphs. In *WWW*, pages 571–580, 2007.

[7] S. Chakrabarti, A. Pathak, and M. Gupta. Index design and query processing for graph conductance search. *VLDBJ*, 20:445–470, 2010.

[8] D. Fogaras, B. Rácz, K. Csalogány, and T. Sarlós. Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments. *Internet Mathematics*, 2(3):333–358, 2005.

[9] Y. Fujiwara, M. Nakatsuji, T. Yamamuro, H. Shiokawa, and M. Onizuka. Efficient personalized pagerank with accuracy assurance. In *SIGKDD*, pages 15–23, 2012.

[10] M. Gupta, A. Pathak, and S. Chakrabarti. Fast algorithms for top-k personalized pagerank queries. In *WWW*, pages 1225–1226, 2008.

[11] T. H. Haveliwala. Topic-Sensitive PageRank: a Context-Sensitive ranking algorithm for web search. *TKDE*, 15(4):784–796, 2003.

[12] G. Jeh and J. Widom. Scaling personalized web search. In *WWW*, pages 271–279, 2003.

[13] S. Kamvar, T. Haveliwala, C. Manning, and G. Golub. Exploiting the block structure of the web for computing PageRank. Technical report, Stanford University, 2003.

[14] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford University, 1999.

[15] A. Papoulis, S. Pillai, and S. Unnikrishna. *Probability, random variables, and stochastic processes*. McGraw-hill New York, 1965.

[16] A. Pathak, S. Chakrabarti, and M. Gupta. Index design for dynamic personalized PageRank. In *ICDE*, pages 1489–1491, 2008.

[17] M. Richardson and P. Domingos. The intelligent surfer: Probabilistic combination of link and content information in pagerank. In *NIPS*, pages 1441–1448, 2002.

[18] P. Sarkar and A. Moore. Fast nearest-neighbor search in disk-resident graphs. In *SIGKDD*, pages 513–522, 2010.