The VLDB Journal (2011) 20:445–470
DOI 10.1007/s00778-010-0204-8

REGULAR PAPER

Index design and query processing for graph conductance search

Soumen Chakrabarti · Amit Pathak · Manish Gupta

Received: 12 September 2008 / Revised: 24 May 2010 / Accepted: 9 September 2010 / Published online: 16 October 2010

Abstract Graph conductance queries, also known as personalized PageRank and related to random walks with restarts, were originally proposed to assign a hyperlink-based prestige score to Web pages. More general forms of such queries are also very useful for ranking in entity-relation (ER) graphs used to represent relational, XML and hypertext data. Evaluation of PageRank usually involves a global eigen computation. If the graph is even moderately large, interactive response times may not be possible. Recently, the need for interactive PageRank evaluation has increased. The graph may be fully known only when the query is submitted. Browsing actions of the user may change some inputs to the PageRank computation dynamically. In this paper, we describe a system that analyzes query workloads and the ER graph, invests in limited offline indexing, and exploits those indices to achieve essentially *constant-time* query processing, even as the graph size scales. Our techniques—data and query statistics collection, index selection and materialization, and query-time index exploitation—have parallels in the extensive relational query optimization literature, but is applied to supporting novel graph data repositories. We report on experiments with five temporal snapshots of the CITESEER ER graph having 74–702 thousand entity nodes, 0.17–1.16 million word nodes, 0.29–3.26 million edges between entities, and 3.29–32.8 million edges between words and entities. We also used two million actual queries from CITESEER's logs. Queries run 3–4 orders of magnitude faster than whole-graph PageRank, the gap growing with graph size. Index size is smaller than a text index. Ranking accuracy is 94–98% with reference to whole-graph PageRank.

S. Chakrabarti (✉) · A. Pathak · M. Gupta
IIT Bombay, Powai, Mumbai, Maharashtra, India
e-mail: soumen@cse.iitb.ac.in

Keywords Personalized PageRank · Graph conductance · Proximity search in graph databases

1 Introduction

Entity-relationship (ER) graphs with textual or structured attributes form a uniform data model for searching the Web [34,42], relational data [3,6,9], XML documents [4], personal data networks [14], and query-document clickthrough graphs [18].

The uniform graph model has been used for many search and mining applications: answering proximity queries [9,47], authority-based text search in databases [6,9], mining connection subgraphs [22] and centerpiece subgraphs [51], detecting stale Web pages [7], disambiguating entities mentioned in e-mails [41], and detecting link spam [28].

Closer scrutiny shows that most of these approaches use one of two interchangeable notions of graph proximity: conductance in an electrical network [20,22,35] and random walks with restarts [32,53], also called personalized PageRank.

1.1 Motivating applications

Suppose a paper P is submitted to a journal editor J, who must find a reviewer R. A personal information database represents papers, words, authors, e-mails, and organizations as typed nodes in an ER graph, connected by typed relations as shown in Fig. 1. Observe that words are also represented as nodes, as is meta-data, such as "company".

A review request is likely to be accepted if R has written papers P' similar to P, or within a short citation distance from P, and R and J have exchanged many e-mails. These can be encouraged in a "schema-free" fashion by measuring

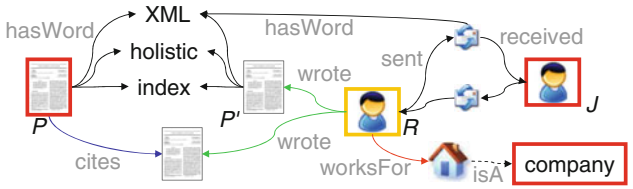

Fig. 1 ER graph conductance search

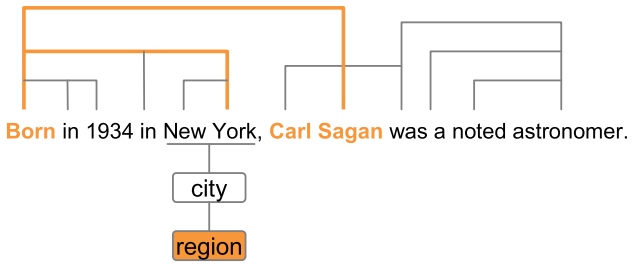

Fig. 2 Searching text with semantic and type annotations

the proximity from the *match nodes* (here, P and J) to the *candidate node* R. Clearly, edges of different types may have very different roles to play in measuring proximity.

A second example comes from searching text with semantic and type annotations (see Fig. 2). The dependency edges connecting the tokens from above come from the CMU Link Parser [49]. Although this is not shown, the edges output by the Link Parser have many different types, depending on the kind of tokens they connect and the purpose of the connections.

In addition to the edges inserted by the Link Parser, a named entity tagger [11,26] may add edges from nodes representing entity types like *city* or *person* to various token spans, e.g., New York *is-a* city. These type nodes may come from a lexical network such as WordNet [15,40].

The combination of links expressing syntactic dependencies among tokens and links connecting tokens to type information gives potent support for semantic search. A query that may be answered with evidence from Fig. 2 is "Where was Carl Sagan born?" posed as a graph proximity query `type=region near "Carl Sagan" born`. A reasonable paradigm for ranking candidate answer tokens is to regard this as a personalized PageRank problem, with teleports to query match words *born* and *Carl Sagan*, and the answer type *region*, induced by *where* in the question. Existing work shows that graph search of this general form can yield good accuracy [6,15].

Yet other notions of graph proximity, based on random walks, have been proposed recently. These include escape probability [52] and hitting and commute times [47]. These new definitions are useful for link prediction and other graph mining tasks. In summary, there is a growing body of recent literature on using random walk processes to define the semantics of proximity queries on graphs, and therefore a growing need to support efficient graph databases to execute such queries.

1.2 Personalized PageRank

We thus see that "personalized" PageRank should be interpreted much more broadly as a form of proximity search in graphs, which can be applied meaningfully at diverse scales: Web pages and hyperlinks, nodes as entities and edges as relations in a personal information space, or nodes as tokens and edges coming from lexical, syntactic, and type annotations. We will now set up some generic, formal definitions of personalized PageRank.

Consider a weighted, directed graph $G = (V, E)$, where nodes represent entities and edges represent relations. Each edge $(u, v) \in E$ in G is associated with a *conductance* $C(v, u)$: this is the probability of a "random surfer" [42] walking from u to v. C is written down as a typically sparse matrix with columns summing to 1. Given the match nodes, we build a multinomial probability vector $r \in \mathbb{R}^{|V|}$ where $r(u) > 0$ only if u is a match node. r is called the *restart* or *teleport* vector and has $\|r\|_1 = 1$.

The random surfer chooses a random match node using the multinomial distribution r. Then, at every step, if the surfer is at u, with probability $1-\alpha$ he *restarts* the walk by invoking r again. With probability α he walks from u to an out-neighbor v using $C(\cdot, u)$. The steady-state visit probability $p_r(u)$ of a node u is its score (p is subscripted with r to remind us that r is the "query"). From the random surfing process, one can arrive at p_r as the infinite series

$$\begin{aligned} p_r &= \alpha C p_r + (1-\alpha) r \\ \therefore\ p_r &= (1-\alpha)(\mathbb{I} - \alpha C)^{-1} r \\ &= (1-\alpha)\left(\sum\nolimits_{k \geq 0} \alpha^k C^k\right) r \end{aligned} \tag{1}$$

The inverse exists, but is never computed explicitly. When computing global PageRank, C and r are fixed. Starting with an initial estimate $\tilde{p}_r^{(0)}$, p_r is estimated using power iterations: $\tilde{p}_r^{(t+1)} \leftarrow \alpha C \tilde{p}_r^{(t)} + (1-\alpha) r$, until $\left\| \tilde{p}_r^{(t+1)} - \tilde{p}_r^{(t)} \right\|$ becomes less than some suitably small $\epsilon_{\text{iter}} > 0$. Here t is the number of iterations.

Personalized PageRank satisfies three desirable properties. First, two nodes are close if there is a high-conductance path between them. Second, two nodes are close if there are *many* paths between them. Third, all else being equal, a node is ranked better if many paths lead to it, a kind of prestige indicator originally used in PageRank [42].

In the example shown in Fig. 1, R should get credit for being close (as in shortest-path distance) to P and J. R should score well if it is connected to J via *many* paths. We should naturally rate R well if R's papers are frequently cited.

Table 1 Notation

$G = (V, E)$	A directed graph with nodes V and edges E
$u, v, w \in V$	Graph nodes representing entities
d	Special dummy node to implement teleport
$C(v, u)$	Probability of making a u to v transition
α	Probability of walk
$1 - \alpha$	Probability of teleport
r	A teleport (multinomial probability) vector
p_r	Personalized PageRank vector for teleport r
ϵ_{iter}	Error tolerance for power iterations
δ_o	Teleport to only one node o
$\text{PPV}_o = p_{\delta_o}$	PageRank vector for teleport δ_o

For future reference, given a fixed node v, if $r(u) = 1$ when $u = v$ and $r(u) = 0$ for $u \neq v$, we write $r = \delta_v$ ("the impulse at node v"), and the resulting PageRank vector p_{δ_v} is called PPV_v, the "personalized PageRank vector for v". Table 1 summarizes the notation used in this paper.

1.3 Quality of results

Balmin et al.[6] gave anecdotal evidence that PageRank queries in ER graphs can give meaningful results. This is confirmed by our experience, especially after edge conductances C were tuned by machine learning techniques [13]. Table 2 shows sample results from a CITESEER snapshot. The top-scoring papers and authors are easily recognized by people familiar with the topic of the query. Additional evidence of the effectiveness of ranking based on properties of random walks have been reported in several recent papers [43,46, 47,52,53]. Not all the applications are about ranking nodes; some consider such novel tasks as link prediction [46] and automatic caption generation for images [43].

1.4 Our contributions

From the preceding discussion, it is clear that a connectivity server [10] with suitable indices for accelerated personalized PageRank computation has widespread applicability.

In this paper, we describe a new system called HUBRANK. The central goal is to build a graph database with suitable indices so that personalized PageRank queries can be answered in (empirically) constant time, irrespective of the size of the data graph, while giving formal guarantees of correct ranking, as per whole-graph PageRank.

We pursue two basic approaches to solve the problem. The main approach proposed here, called **HUBRANKP**, builds upon Berkhin's Bookmark Coloring Algorithm or BCA [8]. BCA has a "spreading activation" flavor: weights are pushed asynchronously along edges, most weights dying down to negligible magnitudes before reaching much of the

Table 2 Top hits for three queries on a snapshot of CITESEER

`type=author near network security`
1. Steven M. Bellovin
2. Eugene H. Spafford
3. Bill Cheswick
4. Matt Bishop
5. Li Gong
6. John Mclean
7. Birgit Pfitzmann
8. Gene Tsudik
`type=author near network congestion control`
1. Van Jacobson
2. Michael J. Karels
3. Sally Floyd
4. Raj Jain
5. Srinivasan Keshav
6. James F. Kurose
7. Don Towsley
8. Songnian Zhou
9. Henning Schulzrinne
`type=paper near shared memory multiprocessor`
1. Weak Ordering - A New Definition
2. An Evaluation of Directory Schemes for Cache Coherence
3. APRIL: A Processor Architecture for Multiprocessing
4. Automatic Translation of FORTRAN Programs to Vector Form
5. Implementing Remote Procedure Calls
6. Scheduler Activations: Effective Kernel Support
7. The MIT Alewife Machine
8. Simple But Effective Techniques for NUMA Memory Management
9. Comparison of Hardware and Software Cache Coherence Schemes
10. Implementing Sequential Consistency In Cache-Based Systems

E-R graph. Early versions of HUBRANKP were reported in WWW 2008 [27] and ICDE 2008 [45].

Apart from standard baselines, we compare HUBRANKP with **HUBRANKD**, based on Jeh and Widom's Hub Decomposition Theorem [32], HUBRANKD identifies an *influence subgraph* for each query that is much smaller than all of G. Thereafter, an interactive computation is performed within the influence subgraph. HUBRANKD was reported in WWW 2007 [12] and is included here not only for completeness, but also because it is not clear *ab initio* which of HUBRANKP and HUBRANKD is superior. While their goals are the same, they are quite distinct algorithms, with different query execution and storage costs. Therefore, it is of interest to compare their performance.

Figure 3 shows the shared layout of the two approaches. Both depend on selecting *hub nodes* $H \subset V$ and building certain indices on H. In both settings, we start with a query processor, and model the typical or average time taken by the

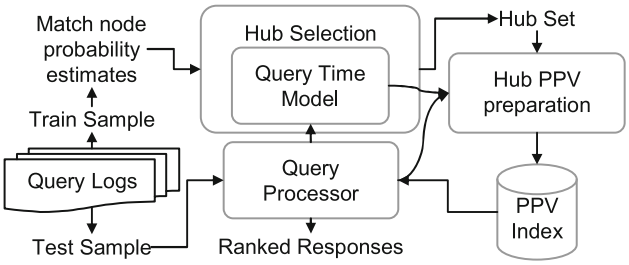

Fig. 3 System outline for HUBRANK and HUBRANKP

query processor over a query workload, as a function of the hub set H selected. Once H is picked, both systems need to prepare and index PPVs for nodes in H.

Section 3 describes HUBRANKD. In Sect. 3.1, we describe query execution, and model its cost. In Sect. 3.2, we give a simple heuristic for selecting H. In Sect. 3.3, we compare HUBRANKD's query speed and accuracy against whole-graph PageRank.

Section 4 describes HUBRANKP. In Sect. 4.4, we describe query execution. In Sect. 4.2, we develop an accurate, yet efficiently estimated predictive model for the runtime performance of the HUBRANKP query processor, as a function of the query, the data graph, and the available index. In Sect. 4.3, we use the performance model and workload statistics to select the hub set, and present experimental results, comparing with HUBRANKD.

We report on experiments with about two million queries from CITESEER's logs and five temporal snapshots of the CITESEER ER graph having 74 to 702 thousand entity nodes, 0.17 to 1.16 million word nodes, 0.29 to 3.26 million edges between entities, and 3.29 to 32.8 million edges between words and entities. We analyze trade-offs between index space, query-processing time, and ranking precision. We have similar experiences with a slice of the US Patent data graph.

Whole-graph PageRank scales linearly with graph size. HUBRANKD runs queries an order of magnitude faster than whole-graph PageRank even for small graphs, and the gap grows indefinitely with graph size. HUBRANKP runs queries 3–4 orders of magnitude faster than whole-graph PageRank, again, the gap growing with graph size. Empirically, our query time is almost *constant*, independent of graph size. The PPV index size is smaller than a regular text index. Per-query RAM footprint for HUBRANKP is 1/15th of HUBRANKD. As a bonus, updating HUBRANKP's state with node browsing actions is as fast as or faster than initiating a new query.

A common issue for both systems is the preparation of the index once H has been selected. By a judicious ordering of nodes in H, and using earlier nodes to assist computation of indices for later nodes, the time to prepare even large hub indices can be made highly sublinear in $|H|$. We present the idea and experimental results in Sect. 5. Concluding remarks are made in Sect. 6.

2 Background and related work

For all but the smallest graphs, global PageRank is too slow for interactive search applications (Table 3). We discuss some existing alternatives to global PageRank computation and argue why none of them fits our needs.

2.1 Linearity and hub decomposition

From (1), observe that p_r is linear in r, and therefore

$$p_{\gamma r} = \gamma p_r \qquad \forall \gamma \in \mathbb{R}, r \in \mathbb{R}^{|V|} \tag{2}$$

and $$p_{r_1+r_2} = p_{r_1} + p_{r_2} \quad \forall r_1, r_2 \in \mathbb{R}^{|V|}. \tag{3}$$

Note that here r, r_1, r_2 need not even be valid multinomial probabilities, but can be *any* real vectors.

Here is an interesting property of p_r that we have not seen commonly mentioned.

Theorem 1 *p_r, which is the solution to the recurrence $p_r = \alpha C p_r + (1-\alpha)r$, also satisfies*

$$p_r = p_{\alpha C r} + (1-\alpha)r.$$

Note that, in $p_{\alpha C r}$, $\alpha C r$ is in the subscript, i.e., $p_{\alpha C r}$ is the PageRank vector corresponding to the input vector $\alpha C r$. $\alpha C r$ is not a valid multinomial teleport probability vector, but we can still plug it in, in place of r, in the definition of PageRank given in (1).

Table 3 Power iteration time (ms) vs. number of entity and word nodes and edges, for queries randomly sampled over origin nodes, and five temporal snapshots of the CiteSeer E-R graph

Year	Entity nodes	Entity edges	Word nodes	Word to entity edges	Avg global Page Rank time (ms)	Standard deviation (ms)	Edges scanned per milli-second
1994	74,223	289,009	171,702	3,289,201	2,505	724	4,711
1996	177,208	755,762	341,437	8,530,197	7,159	1,765	4,494
1998	319,608	1,430,630	562,420	15,661,128	15,048	4,016	4,558
2000	470,028	2,157,541	792,794	23,309,842	27,746	6,475	4,583
Full	702,406	3,261,041	1,163,818	32,762,013	39,586	8,170	4,667

Proof Start from the rhs:

$$\begin{aligned}(1-\alpha)r + p_{\alpha Cr} &= (1-\alpha)r \\ &+(1-\alpha)\left(\sum_{k\geq 0}\alpha^k C^k\right)(\alpha Cr) \quad \text{using (1)} \\ &= (1-\alpha)\left(\mathbb{I}+\sum_{k\geq 1}\alpha^k C^k\right)r = p_r,\end{aligned}$$

where we have used (3) twice. (An easy mnemonic for this equality is to start with $p_r = (1-\alpha)r + \alpha C p_r$, and "push down" the αC into the teleport argument of p.) □

A simple corollary to Theorem 1 is the following hub decomposition result, shown earlier by Jeh and Widom [32] from first principles.

Theorem 2 (Hub Decomposition)

$$p_{\delta_u} = \alpha \sum_{(u,v)\in E} C(v,u)p_{\delta_v} + (1-\alpha)\delta_u.$$

Proof Using $r = \delta_u$ in Theorem 1, we just need to show that $p_{\alpha C\delta_u}$, which is $\alpha p_{C\delta_u}$ using (2), is equal to $\alpha\sum_{(u,v)\in E} C(v,u)p_{\delta_v}$, i.e., we need to show that

$$p_{C\delta_u} = \sum_{(u,v)\in E} C(v,u)p_{\delta_v} \overset{(3)}{=} p_{\sum_{(u,v)\in E} C(v,u)\delta_v},$$

using (3). Therefore, it suffices to see that both $C\delta_u$ and $\sum_{(u,v)\in E} C(v,u)\delta_v$ are the uth column of C. □

The hub decomposition theorem says that if all out-neighbors of node u have known PPVs, the PPV of u can be computed easily. The result can also be written in the combined matrix form

$$Q = \alpha QC + (1-\alpha)\mathbb{I}, \tag{4}$$

where the uth column of Q is PPV_u and $\mathbb{I}$ is the $|V|\times|V|$ identity matrix. Note that, unlike in (1), Q multiples C from the *left*.

Hub selection Jeh and Widom went on to give a recipe for computing p_r for an arbitrary r, given that PPV_h was precomputed and stored for a fraction of nodes $h \in H$. They thus laid the groundwork for personalization, but absent was a treatment of how $H \subset V$ ought to be picked. Their only suggested heuristic was to pick "large PageRank" nodes into H, because these are easily reached from other nodes. In particular, they did not consider that nodes can behave very differently in queries, e.g., because some represent words and others entities. They also did not consider how a query workload can be used to select a better H. This is our focus.

2.2 ObejctRank and BinRank

In ObjectRank [6], the PPVs of all word nodes are precomputed and cached. Only keyword match node are allowed. Therefore, the response to a query can be computed as a simple Fagin merge [21] of PPVs of words in the match set, without any graph computation at query time.

ObjectRank seeks to make keyword query-processing interactive, but its preprocessing costs quickly get out of hand with increasing graph and corpus vocabulary size. Precomputing PPVs for all 562,000 words in our testbed takes an estimated 22,000 CPU-hours. Unsurprisingly, ObjectRank does not really precompute PPVs for *all* words, but maintains a cache of word PPVs. In case a user query "misses" in this cache, a message of the following form results[1]

```
Top 20 results for keywords: euler lagrange
[Message: INDEX NOT FOUND]
Sorry. The answer to your query has not been
precomputed and stored in our system yet. It
would become available in the near future.
Thank you for your patience.
```

BinRank [30], published after most of the building blocks [12,27,45] in this paper were published, attacked this key bottleneck in ObjectRank. They used the property that growing the teleport ("base") set does not thin the set of nodes that have non-negligible scores. They precomputed a partitioned clustering ("binning") of all words in the corpus, where the similarity measure used in the clustering was based on cooccurrence of words in object nodes. For each bin, they precomputed and stored a subgraph similar to our notion of a influence subgraph. At query time, a bin that contains all query words and has a small influence subgraph is chosen, the graph loaded into RAM, and ObjectRank scores computed in the restricted graph. BinRank reduced the RAM footprint compared to HubRankD, but as we shall see, HubRankP [27,45] is a remarkable improvement beyond HubRankD, smoothly trading off between query time and index space and preparation time, and giving formal guarantees on accuracy.

2.3 Asynchronous PageRank

Abiteboul et al.[1] described OPIC, an algorithm to compute PageRank with a fixed teleport vector, while the graph is changing and a crawler is incrementally acquiring parts of the graph. OPIC proceeds by assigning some initial "cash" to each node and "pushes" cash asynchronously along edges. OPIC does not handle dynamic updates to teleport r: "One

[1] The output was captured from http://teriyaki.ucsd.edu:9099/examples/jsp/objrank/objectRank05.jsp on 2006/11/12. The same demo is on http://db.ucsd.edu/BibObjectRank/main05new.html at this time.

may also want to bias the ranking of answers based on the interest of users. Such interesting aspects are ignored here." OPIC does not propose or build indexes offline to speed up query-time PageRank computation. McSherry [39] gave a related asynchronous update scheme and gave effective heuristics for scheduling the "cash propagation", but he did not use any indexes; neither did Sarkar et al.'s algorithm [47]. The most recent asynchronous update algorithm, one on which we base our current work, is Berkhin's BCA ("bookmark coloring algorithm") [8].

Proposition 3 *A converging approximation $\hat{p}_r$ to the sum in (1) can be built as follows:*

1: $q \leftarrow r,\ \hat{p}_r \leftarrow \mathbf{0}$
2: **while** $\|q\|_1$ is large **do**
3: $\quad \hat{p}_r \leftarrow \hat{p}_r + (1-\alpha)q$
4: $\quad q \leftarrow \alpha C q$

Proof The result is clear from the expression of p_r as the infinite series $p_r = (1-\alpha)\left(\sum_{k\geq 0} \alpha^k C^k\right) r$. □

Because columns of C sum to 1, $\|Cq\|_1 = \|q\|_1$, $\left\|q^{(t+1)}\right\|_1 = \alpha \left\|q^{(t)}\right\|_1$, and fast convergence is guaranteed. We can replace the "synchronous" update in Step 4 of the whole of q with an "asynchronous push" from some node u to its out-neighbors v. This let Berkhin use a sparse form of C.

Proposition 4 *A converging approximation $\hat{p}_r$ to the series $p_r = (1-\alpha)\left(\sum_{k\geq 0} \alpha^k C^k\right) r$ can be computed using the following code:*

1: $q \leftarrow r,\ \hat{p}_r \leftarrow \mathbf{0}$
2: **while** $\|q\|_1$ is large **do**
3: $\quad$ pick a node u such that $q(u) > 0$
4: $\quad \hat{q} \leftarrow q(u),\ q(u) \leftarrow 0$
5: $\quad \hat{p}_r(u) \leftarrow \hat{p}_r(u) + (1-\alpha)\hat{q}$
6: $\quad$ **for** each out-neighbor v of u **do**
7: $\quad\quad q(v) \leftarrow q(v) + \alpha C(v,u)\hat{q}$

The proof of the above statement is a special case of the proof of Proposition 7, so we omit it.

Berkhin also proposed to use PPVs of hub nodes to trade-off between disk space and query time, but he left two important questions unanswered.

- There was no known model that could predict the execution time of a specific query, based on the choice of hubs and other system parameters (such as convergence tolerances).
- Lacking a predictive performance model, he proposed a generic hub selection strategy **LPR** similar to Jeh and Widom: select "Large-PageRank" nodes as hubs.

While the LPR policy might be acceptable for the Web graph, it performs poorly with diverse node and edge types. For example, in our setting, word nodes are many in number, and therefore each has small global (query-independent) PageRank, but they are vitally important to include into the hub set, because most trails begin with keyword queries. But neither words nor entities by themselves can form as good a hub set as a judicious mix of both, found by exploiting workloads of sample match node sets.

Our second proposal in this paper, HubRankP, uses BCA as a foundation to implement very fast incremental updates to p_r as r changes in a sparse fashion.

2.4 Experimental testbed

In the interest of continuity, we will outline experimental results as we present ideas and algorithms. In this section, we summarize our experimental testbed and some baseline performance numbers.

2.4.1 Hardware and software

HubRankD and HubRankP were written in Java (64-bit JDK1.5) and run on a 4-CPU 2.2 GHz Opteron server with 8 GB RAM and U320 disks. Baseline PageRank used $\alpha = 0.8$; power iterations were stopped when $\left\|\tilde{p}_r^{(t+1)} - \tilde{p}_r^{(t)}\right\|$ dropped below $\epsilon_{\text{iter}} = 10^{-6}$.

2.4.2 The ER data graphs

The full CiteSeer corpus has 1,163,818 words over 702,406 entity nodes and 3,261,041 edges between entities. To get naturally scaling graphs, we took the snapshots up to years 1994, 1996, 1998, 2000, as well as the full graph. Node and edge statistics for the data slices are shown in Table 3. Except in scaling studies, all the experimental numbers reported are for the 1994 snapshot.

Lucene (http://lucene.apache.org) text indices took 55, 139, 259, 378, 540 MB on disk, respectively, on these snapshots. The full CiteSeer graph skeleton occupied only 100 MB RAM. The samples were smaller, so HubRankD and HubRankP used an in-memory conductance graph with all other metadata kept on disk. Recent work [19] has enabled using intelligent multilevel memory hierarchy for graph queries.

Our system can process graphs that are orders of magnitude larger, but we could not locate an object graph with query logs that are so critical to our success. For some object graphs, we sampled the distribution of words in the corpus itself as a crude "query log". The behavior of our algorithms remained broadly similar.

2.4.3 Whole-graph PageRank

Empirically, for a fixed ϵ, a constant number of power iterations (typically between 50 and 70 for us, consistent

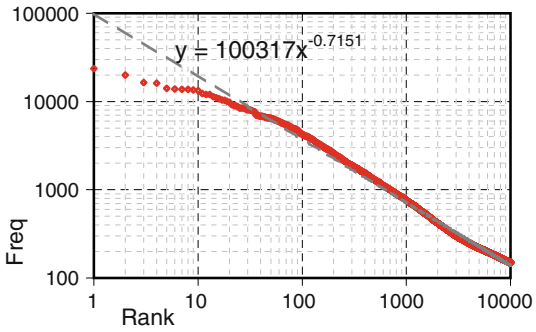

Fig. 4 Typical Zipfian distribution of word frequencies over almost two million queries

with prior experience [33]) are needed, each iteration taking $O(|V|+|E|)$ time. This is because the PageRank loop looks like this:

1: $p_{\text{next}} \leftarrow \mathbf{0}$
2: **for** each edge $\langle u, v, C(v,u)\rangle$ **do**
3: $\quad p_{\text{next}}(v) \leftarrow p_{\text{next}} + p_{\text{cur}}(u)\, C(v,u)$

where edges can be read in arbitrary order from disk. Because the amount of CPU work inside the loop is very small, access to an edge and its conductance must be very fast.

If u, v *and* C are accessed directly from three native Java arrays (`int[], int[], double[]`), global PageRank can scan around 20 million edges per second on our hardware, which is reasonable for a JVM. However, per-query modifications to G necessitates some extent of programming abstraction involving edge iterators and a class to represent edges with conductances, rather than reading these from native arrays directly. These, coupled with heap management overheads, reduces our scan rate to around five million edges per second. Baseline whole-graph PageRank times are shown in Table 3. We believe these speeds can be boosted by a factor of 2–3 by coding in C++.

Note that, for a given graph implementation, the time to access an edge is the same across all algorithms. Therefore, we can also compare two algorithms in terms of the number of edges they access, which is platform-independent. In some cases, we also report this performance measure for completeness.

2.4.4 The query log

We obtained 1.9 million queries from CiteSeer, collected over 36 days from 21st Aug to 25 Sep, 2005 with an average of 2.68 words per query. Word frequencies are distributed in a familiar Zipfian manner, as shown in Fig. 4. We sampled 10,000 test queries from the first 100,000 queries, while all but the first 100,000 queries were used to train and tune our indices. This sampling procedure (unlike uniform random sampling) made sure that we are not benefiting from just the short-term memory of splitting a query session with shared words into the training and test set. Choosing the (small) test set to be chronologically *earlier* than the (larger) training set further ensures that the benefits of the hub index are measured based only on long-term and stationary word statistics. We chose several different test samples of size 10,000 to make sure our observations are stable.

2.4.5 Workload model

In HubRank, a query is modeled as an unordered set of words. Usually, words are represented as nodes; therefore, each query can be thought of as a set of match nodes. A query word can be any word in the vocabulary of the text associated with nodes. (If phrases are detected in the corpus in advance [38], these phrases can be used as indexing units as well.)

To completely characterize a workload, we have to estimate a distribution over all subsets of the vocabulary. Computationally, capturing all dependencies between all words is quite impractical. Luckily, most queries are very short, typically, 1–3 words. In this section, we will build a unigram model, i.e., estimate the marginal probability for each single word w, which we denote $queryProb(w)$. These estimates will be used to estimate the average cost of query in Subsect. 3.1.4 and Sect. 4.2, which will then be used to design the hub index in Sects. 3.2 and 4.3, respectively.

The problem with estimating $queryProb(w)$ is that in most search engines, query word frequencies are highly skewed. A sizable fraction of queries are never repeated. Even if we use a large training set, a separately sampled test set will always contain words we never saw in the training set. Each such query may be rare, but collectively, many such queries will happen. If we do not provision for these low-probability word events, overall query execution performance will suffer.

This is a standard problem in language modeling [38], and a variety of smoothing schemes are known. We use the well-known Lidstone smoother:

$$queryProb(w) = \frac{queryCount(w) + \ell}{\sum_{w'} queryCount\,(w') + \ell}, \tag{5}$$

where $queryCount(w)$ is simply the raw count of w in the training set and $0 < \ell \leq 1$ is a parameter to be set via cross-validation.

To tune ℓ, we randomly split the workload into partitions W_1 and W_2, estimate $queryProb(w)$ using W_1, and estimate the probability of W_2 as

$$\sum_{w \in W_2} queryCount_{W_2}(w) \log\left(queryProb_{W_1,\ell}(w)\right),$$

and we maximize over tuning parameter ℓ:

$$\arg\max_{\ell} \sum_{w \in W_2} queryCount_{W_2}(w) \log \left(queryProb_{W_1,\ell}(w)\right).$$

using a grid search. For robustness we can make multiple random W_1, W_2 splits.

For HubRankP, match nodes can correspond to both words and entities, but as long as we have statistics for users issuing word queries and clicking on entity nodes, the problem remains the same. A match node set M can be any subset of V.

Suppose the query time with M as input is denoted QueryTime(M). Then the expected query time over a workload is

$$\sum_{M} \Pr(M)\, \text{QueryTime}(M).$$

A full characterization of $\Pr(M)$ will involve estimating $2^{|V|}$ probabilities. Luckily, most Ms are very small and most $\Pr(M)$s are vanishingly small. A browsing action corresponds to a singleton match set $M = \{u\}$. A keyword search step also typically results in a very small match set because most queries are only 1–3 words long [48].

Therefore, in the interest of computational feasibility, similar to word probability estimation in language modeling [38], we limit ourselves to estimating single-node marginals $queryProb(u)$, which is the probability that a query drawn randomly from the universe of queries has a match set M containing u. Using $queryProb(\cdot)$, we approximate the expected query-processing time as:

$$\sum_{u \in V} queryProb(u)\, \text{QueryTime}(\{u\})$$

i.e., the match set is the singleton node u.

If it is found desirable to extend the query distribution beyond a multinomial on single words, one can extend market basket techniques [29] to build a dictionary of highly correlated word sets, and then estimate probabilities of seeing these sets in queries. We leave this for possible future work.

2.4.6 Evaluating scores and rankings

Although it takes much more time, running whole-graph personalized PageRank gives us the gold-standard scores and ranking among nodes. We compare HubRankD and HubRankP scores and ranks against the PageRank reference as follows.

L_1 error If p_r is the true full-precision personalized PageRank, and we estimate $\hat{p}_r$, then $\left\| \hat{p}_r - p_r \right\|_1$ is a reasonable first health-check, but it does not guarantee *ranking* accuracy.

Precision at k p_r induces a "perfect" ranking on all nodes v, while $\hat{p}_r$ induces an approximate ranking. Let the respective top-k sets be T_k and $\hat{T}_k$. Then the precision at k is defined as $|T_k \cap \hat{T}_k|/k \in [0, 1]$. Clipping at k is reasonable, because, in applications, users are generally not adversely affected by erroneous ranking lower in the ranked list.

Relative average goodness (RAG) at k Precision can be excessively severe. In many real-life social networks, nearties in PageRanks are common.

$$\text{RAG}_k = \frac{\sum_{v \in \hat{T}_k} p_r(v)}{\sum_{v \in T_k} p_r(v)} \in [0, 1]$$

rewards the ranking algorithm if the *true scores* of $\hat{T}_k$ are large (note that $\hat{p}_r(v)$ is not used). RAG can be overly lenient.

Kendall's τ Let exact and approximate node scores be denoted by $S_k(v)$ and $\hat{S}_k(v)$, respectively, where the scores are forced to zero if $v \notin T_k$ and $v \notin \hat{T}_k$. A node pair $v, w \in T_k \cup \hat{T}_k$ is *concordant* if $(S_k(v) - S_k(w))\left(\hat{S}_k(v) - \hat{S}_k(w)\right) > 0$, and *discordant* if it is < 0. It is an *exact-tie* if $S_k(v) = S_k(w)$ and is an *approximate tie* if $\hat{S}_k(v) = \hat{S}_k(w)$. If there are c, d, e and a such pairs, respectively, and m pairs overall in $T_k \cup \hat{T}_k$, then Kendall's τ (at rank k) is defined as

$$\tau_k = \frac{c - d}{\sqrt{(m - e)(m - a)}} \in [-1, 1].$$

Other criteria Information retrieval systems are evaluated using test queries, each having a set of human-labeled relevant and irrelevant documents (see http://trec.nist.gov/). Typically, there is no total order of the corpus for each query. In such settings mean average precision (MAP), mean reciprocal rank (MRR), or normalized discounted cumulative gain (NDCG) are appropriate [31]. In contrast, we have a reference ranking from the definition of personalized PageRank, but no absolute relevance judgments. Therefore, we use criteria that are based on comparing scores and permutations.

3 HubRankD

For fast query processing, both HubRankD and HubRankP depend on the fact that r is sparse, i.e., teleport initially reaches only a few nodes, which we called the "match nodes". The two algorithms limit computations to a small subgraph reachable from the match nodes. HubRankD does this in a heuristic manner, whereas HubRankP uses the more principled BCA approach. To keep the distinction clear, we call the subgraph the *influence subgraph* in case of HubRankD and the *active subgraph* in case of HubRankP.

1: **input:** match nodes W, hub nodes H, threshold $\epsilon_{\text{trim}} > 0$
2: let *frontier* be a max priority queue of records $\langle v, s\rangle$ where v is a node and s is the priority score
3: **for** each match node w **do**
4: insert $\langle w, 1\rangle$ into *frontier*
5: **while** *frontier* is not empty **do**
6: remove $\langle u, s\rangle$ from *frontier*
7: **if** $u \notin I$ **then**
8: add u to I
9: **if** $u \notin H$ and $s > \epsilon_{\text{trim}}$ **then**
10: **for** each child v of u **do**
11: add $\langle v, s\,\alpha\,C(v,u)\rangle$ to *frontier*
12: **return** influence subgraph nodes $I \subset V$

Fig. 5 Query-time collection of influence subgraph

3.1 Query execution algorithm and cost

Suppose u is a match node, i.e., $r(u) > 0$. Suppose we knew PPV_u for all such u's. Then, we can quickly compute the scores of all nodes in response to query r as $\sum_u r(u)\,\text{PPV}_u$, similar to ObjectRank. Theorem 2 says that, even if PPV_u were not known at query time, if all out-neighbors of node u have known PPVs, the PPV of u can be computed easily. HubRankD extends this to the case where even the direct out-neighbors of u do not have known PPVs.

3.1.1 Expansion to influence subgraph

After receiving a query, the first step of HubRankD is to expand out from the match nodes u until a hub node h with a known PPV is hit or (an approximation to) the conductance from u to the periphery node v dwindles to a very small value. The intuition is that nodes like v have little influence on PPV_u. So the exact PPV_v we use is unimportant, and we can just use δ_v, say.

The algorithm for collecting the influence graph is shown in Fig. 5. There is no formal property associated with the influence subgraph nodes I, because we are approximating the conductance from a teleport node u to an influence node v with the single largest conductance path from u to v. Multiple paths are not taken into account. ϵ_{trim} is a system parameter that calls off further expansion once we are sufficiently far from match nodes.

3.1.2 PPV *clipping and loading heuristics*

We describe how PPVs are computed for hub nodes in Sect. 5. Fogaras et al.[24] show that to achieve high confidence in arranging nodes by personalized PageRank, any index must have $\Omega(|H|\,|V|)$ bits in the worst case. In realistic graphs and practical applications, PPV elements are extremely skewed, query words are Zipfian, and the graph often shows power-law degree distribution. This allows practical systems like ObjectRank [6] to *clip* PPVs, i.e., remove any $\text{PPV}_h(u) <$

1: **input:** influence nodes I, nodes $M \subset I$ with missing PPVs, hub nodes $H \cap I$, distant nodes $D \subset I$, tolerance ϵ_{iter}
2: for $h \in H \cap I$ initialize PPV_h from PPV index and keep fixed
3: for $d \in D$ initialize $\text{PPV}_d = \delta_d$ and keep fixed
4: for $m \in M$ initialize $\text{PPV}_m^{(0)} = \delta_m$
5: **for** $t = 1, 2, \ldots$ **do**
6: **for** each $m \in M$ **do**
7: $\text{PPV}_m^{(t)} \leftarrow (1-\alpha)\delta_m$
8: **for** $(m, v) \in E$ **do**
9: read PPV_v as one of $\begin{cases}\text{PPV}_h\\ \text{PPV}_d\\ \text{PPV}_v^{(t-1)}\end{cases}$ as appropriate
10: $\text{PPV}_m^{(t)} \leftarrow \text{PPV}_m^{(t)} + \alpha C(v,m)\,\text{PPV}_v$ {Theorem 2}
11: **if** $\max_m \|\,\text{PPV}_m^{(t)} - \text{PPV}_m^{(t-1)}\,\| < \epsilon_{\text{iter}}$ **then**
12: **return**
13: **return** estimate of PPV_m for all $m \in M$

Fig. 6 Iterative computation of missing PPVs

ϵ_{clip} for some system parameter $\epsilon_{\text{clip}} > 0$, before storing it in the index. This makes the PPVs much smaller. More PPVs fit in a RAM cache of a given size and queries can be executed faster. Obviously, personalized PageRank scores will become more and more inaccurate as ϵ_{clip} increases. As a result, the ranking induced by HubRank will also deviate from the gold standard, and this can be measured in various ways, as described in Subsect. 2.4.6.

ϵ_{clip} is usually chosen conservatively such that ranking accuracy is unlikely to suffer for *any* query. (Typically, testing the system with a few values of ϵ_{clip} suffices; see Fig. 12.) However, when the algorithm in Fig. 5 is executed for a specific set of match nodes, if u is far away from the match nodes, then s is very small. In such cases, we can load only a part of PPV_u, as follows. We keep all clipped PPV_h sorted in decreasing order of $\text{PPV}_h(v)$ and load up to the last element that is larger than ϵ_{trim}/s. The intuition is that $\text{PPV}_h(v)$ can significantly affect the answer only if $s\,\text{PPV}_h(v) > \epsilon_{\text{trim}}$. As we shall see in Sect. 3.3, this additional adaptive clipping does not affect ranking accuracy in practice, but improves query-processing time.

3.1.3 Computing missing PPVs

After executing Fig. 5, we have an influence graph I where

- The PPVs at the nodes in $I \cap H$ can be read from the PPV database precomputed at indexing time.
- The PPVs at some other nodes $v \notin H$ are fixed to δ_v because they are likely to have small influence on the query result. We call these *distant* nodes $D \subset I$.
- The remaining nodes have *missing* PPVs and are denoted $M \subset V$. We have to find these PPV_ms.

Figure 6 shows a customized power iteration pseudocode.

Proposition 5 *If, in the algorithm shown in Fig. 6, the termination condition is*

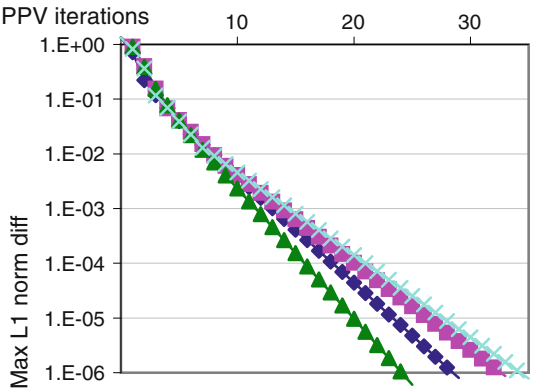

Fig. 7 Fast convergence of missing PPVs. Each line is a different query

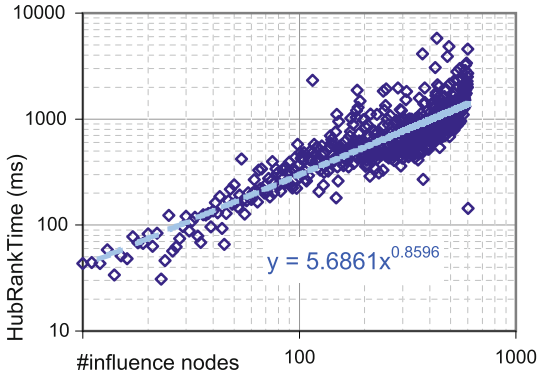

Fig. 8 Over 2–3 orders of magnitude, the time for iterative PPV estimation is almost linear in the number of influence nodes

$$\forall m \in M: \left\| \mathrm{PPV}_m^{(t)} - \mathrm{PPV}_m^{(t-1)} \right\| < \epsilon_{iter}$$

for some tolerance $\epsilon_{iter} > 0$*, then the algorithm converges, giving a definite value for all* PPV_m*s.*

Proof Consider (4) again. Let PPV_u be the uth column in $Q \in \mathbb{R}^{|V|\times|V|}$, and let a specific row of Q (corresponding to a fixed node $w \in V$, say) be q. Then PPV iterations amount to solving for q the recurrence $q = \alpha\, q\, C + (1-\alpha)\delta_w^\top$, except that q is partitioned into q_M, the missing elements and q_K, the known elements (from hub and distant nodes). Let C be correspondingly partitioned into $\begin{bmatrix} C_{MM} & C_{MK} \\ C_{KM} & C_{KK} \end{bmatrix}$. As far as our PPV iterations go, because we never look beyond nodes in K, only $M \to K$ and $M \to M$ edges matter; thus, we are looking for a solution to

$$q_M = \alpha\, q_M\, C_{MM} + \alpha\, q_K\, C_{KM} + \mathrm{const}_{1\times|M|}$$

but $\alpha\, q_K\, C_{KM}$ is a fixed row vector as well, so the recurrence simplifies into $q_M = \alpha\, q_M\, C_{MM} + c$, where c is some fixed $1 \times |M|$ row vector. Because $\alpha\, C_{MM}$ is strictly substochastic, $(\mathbb{I} - \alpha\, C_{MM})^{-1}$ exists and so there is a unique solution for q_M. Now, we can adapt Jeh and Widom's proof of convergence of their "basic dynamic programming" method to show that the algorithm in Fig. 6 will converge to these values. □

For four representative queries (one line per query), Fig. 7 shows that the worst L_1 change in any missing PPV indeed goes down exponentially with iterations. Unfortunately, q_M is not *guaranteed* to be statistically meaningful (e.g., unbiased or even nonnegative). This makes HubRankP, based on BCA, a better choice, at least as far as the theory goes.

3.1.4 Query cost

HubRankD's query execution cost has three parts:

1. Building the influence subgraph and loading known PPVs.
2. Iteratively computing missing PPVs.
3. Merging teleport node PPVs and reporting top-K result nodes.

The iterative computation shown in Fig. 6 dominates query-processing time, especially if PPVs are stored truncated as described in Subsect. 3.1.2. To a first approximation, the number of nodes in the influence graph strongly affects iteration time. Regressing on a log–log plot (shown in Fig. 8) gives a slightly sublinear fit. This very crude estimate is accurate to within a factor of 2 for a vast majority of queries.

3.2 Hub selection

Given the dependence of query time on the size of the influence graph, a very reasonable objective, while selecting H, is to try to minimize the size of the influence subgraph, summed (equivalently, averaged) over queries in a representative workload. However, this can be shown to be NP-hard.

Proposition 6 *The NP-complete independent set problem is polynomial-time reducible to the problem of finding an optimal hub set.*

Proof The independent set problem is as follows. Given an undirected graph $G = (V, E)$ and a number $0 < k < |V|$, decide whether there is a subset $K \subset V$ of size at least k such that no pair of nodes in K are connected by an edge in G. Given an instance of the independent set problem, we create a directed graph for HubRankD by bidirecting all edges. Then, we ask for the best hub set of size at most $|V| - k$. Suppose G had an independent set K of size at least k. Then the hub set selection algorithm is free to choose $H = V \setminus K$. Any query can now be executed by directly reading off a PPV, or a single step combination of PPVs as in Theorem 2. In contrast, if there is no independent set of size at least k, and H must be of size at most $|V| - k$, there will be some edge (u, v) such that $u \notin H$ and $v \notin H$, and there is some query teleport r for which the iterative algorithm in Fig. 6

```
1: input: queryProb(w) for each word w, ER graph
2: initialize a score map score(u) for word and entity nodes u
3: for each word node w do
4:     let frontier = {w}
5:     let priority(w) = queryProb(w)
6:     create an empty set of visited nodes
7:     while frontier ≠ ∅ do
8:         remove some u from frontier and mark visited
9:         accumulate priority(u) to score(u)
10:        for each visited neighbor v do
11:            accumulate priority(u) α C(v, u) to score(v)
12:        for each unvisited neighbor v do
13:            let priority(v) = priority(u) α C(v, u)
14:            add v to frontier
15: sort word and entity nodes by decreasing score(u)
16: return merit order on word and entity nodes
```

Fig. 9 Greedy estimation of a measure of merit to include each node into the hub set

must be run. We can make the convergence criterion very stringent (small ϵ_{iter} in Proposition 5) so that the resulting query time is much larger than simple PPV lookups, in fact, so large that we can detect a difference in average query time over a workload of queries, each teleporting to one node in G. □

Since the estimates of query-processing times are anyway approximate, and so are the estimates of disk space needed to store PPVs, the likely non-existence of a polynomial-time algorithm is not too disturbing. In fact, on large ER graphs, any algorithm that takes much more than linear time may not be acceptable. We will use a simple greedy heuristic: we will order word and entity nodes in decreasing order of an intuitive and easy-to-compute merit score, then pick a suitable prefix of the merit list. The algorithm will closely mirror the expansion to the influence subgraph shown in Fig. 5, except that the query workload will be taken into account. The algorithm is shown in Fig. 9. Once the merit list is set, we go down the list computing PPVs (see Sect. 5) and stop when we run out of index space.

There are two limitations of the above approach. First, the merit of a node u depends not only on the degree to which activation from queries reaches u, but also the "hinterland" behind u that would not enter the influence graph if u were made a hub node. The latter factor is ignored by HubRankD. Second, we do not account for the space occupied by PPVs while constructing the merit list; ideally, we should do a cost-benefit analysis. We fix both these problems in HubRankP and experimentally show that these fixes result in considerable improvements.

3.3 Performance

Choice of hub nodes For keyword queries, teleport is strongest at word nodes and then diffuses out to entities with a loss incurred at every step. Therefore, it may appear that the

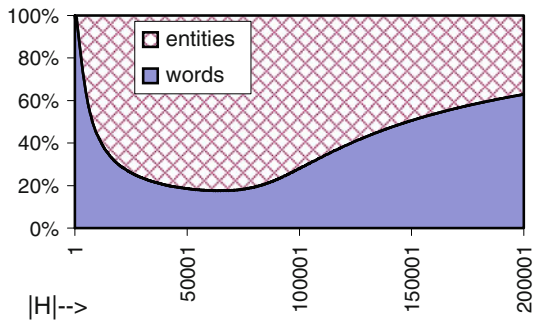

Fig. 10 For reasonable hub set sizes, entity nodes are highly desirable compared with word nodes; the best case is a nontrivial mix

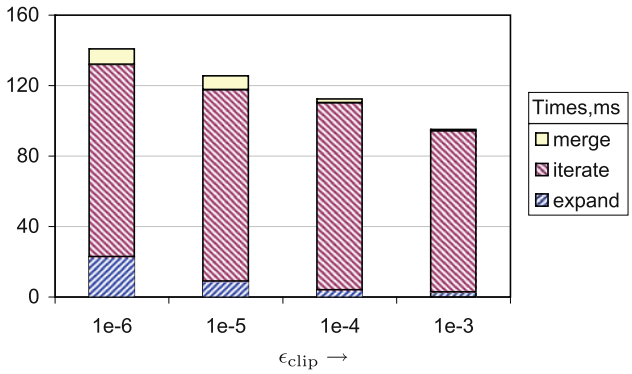

Fig. 11 Effect of storing clipped PPVs on query speed ($|H| = 16{,}000$)

originating word nodes have all the advantage in ranking highest in the merit list. However, the correct intuition is that queries about a link-cluster in the graph will share a theme but not necessarily words. Over many queries, these individual words may not float to the top, but entity nodes at the confluence of many short paths from thematic words will.

This is confirmed in Fig. 10. The order returned by the algorithm in Fig. 9 is a nontrivial mix. Words do crowd the top ranks but soon they are overtaken by entity nodes; in fact, the fraction of words steadily dwindles until nearly all entity nodes are exhausted.

Effect of PPV clipping PPVs are initially stored in the disk PPV cache so that PPV elements less than ϵ_{clip} are clipped and discarded. We will vary ϵ_{clip} between 10^{-6} and 10^{-3}. Now queries are run with a conservative value of $\epsilon_{\text{trim}} = 10^{-6}$, effectively disabling trimmed loading. Figure 11 shows the effect of ϵ_{clip} on average query execution time, broken down into the main subroutines. Larger values of ϵ_{clip} (more aggressive clipping) saves time during the influence graph expansion phase (which includes PPV loading—see Subsect. 3.1.4) as well as the iterative phase, because PPVs are represented as sparse vectors and increasing ϵ_{clip} increases sparsity.

Figure 12 shows the effect of clipping PPVs on ranking accuracy, with $k = 20$. RAG, being very forgiving, remains essentially 1 throughout. Precision and Kendall's

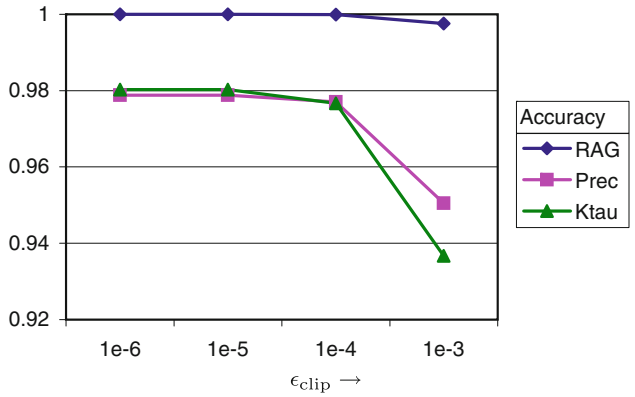

Fig. 12 Effect of storing clipped PPVs on ranking accuracy ($|H| = 16{,}000$)

Table 4 Total space needed to store PPVs (in megabytes) as a function of $|H|$ and ϵ_{clip}. A Lucene text index for this data set takes 55 MB

		$\epsilon_{clip} \rightarrow$ 10^{-6}	10^{-5}	10^{-4}	10^{-3}
$\lvert H\rvert$	5,000	293	106	26	4
$\downarrow$	10,000	438	155	40	7
	15,000	561	196	51	10
	20,000	679	235	63	12
	25,000	793	272	73	15
	30,000	910	311	84	18
	35,000	1,020	347	95	21

τ are around 0.98 up to $\epsilon_{clip} = 10^{-4}$, but further clipping makes matters worse.

Table 4 shows the payload size of the PPV index on disk, against the two design choices of $|H|$ and ϵ_{clip}. Each sparse record in a PPV is assumed to be one 4-byte `int` for the node ID and one 8-byte `double` for the PageRank value. As a reference, for this data set (1994) a standard Lucene text index is 55 MB large. For this data set, $\epsilon_{clip} = 10^{-4}$ is an excellent compromise between PPV index size, query execution speed, and ranking accuracy.

Here, we computed the table entries after building the PPV index, but Subsect. 4.3.3 will show how the sizes can be estimated quickly *without* materializing the PPVs first. Armed with this table, we should pick the smallest ϵ_{clip} consistent with our storage budget.

Effect of trimmed PPV loading Next, we will measure the effect of varying ϵ_{trim} on query speed while holding ϵ_{clip} and $|H|$ (therefore, the size of the PPV index) fixed. ϵ_{trim} affects all stages of query processing:

– Large ϵ_{trim} quickly terminates the influence graph expansion show in Fig. 5.

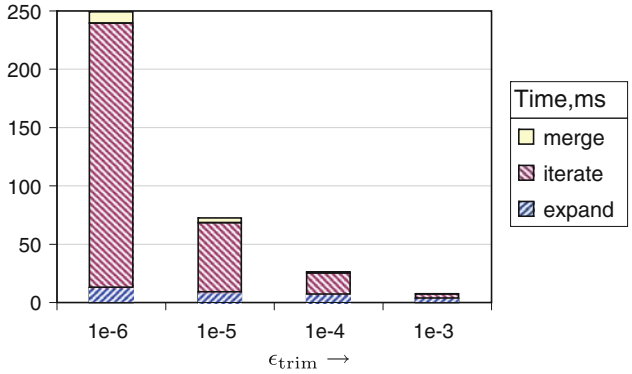

Fig. 13 Effect of ϵ_{trim} on query time, broken down into query-processing stages

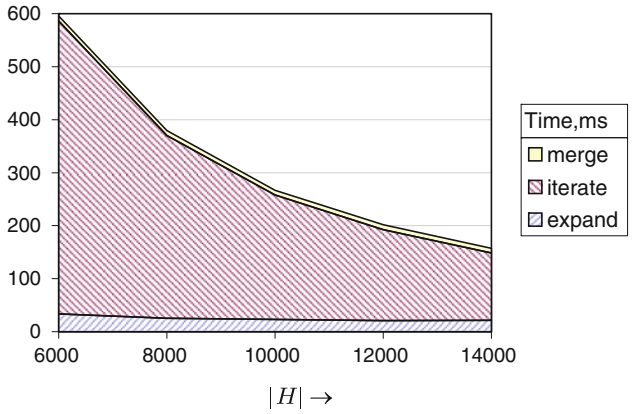

Fig. 14 Breakup of HubRank running time

– Large ϵ_{trim} leads to early abandonment of PPV loading (Subsect. 3.1.2). However, most PPVs are only hundreds to thousands of bytes long, so this may not save much in terms of disk IO.
– However, loading larger PPVs lead to the PPVs in the iteration step (Fig. 6) being less sparse, and this leads to a dramatic increase in the time required for the iteration step, and a smaller increase in the time to merge the final PPVs into a ranking. This is the dominant reason why large ϵ_{trim} leads to much faster query execution.

These observations are summarized in Fig. 13.

Query time breakup Figure 14 shows the time taken by various stages of HubRankD's query-processing algorithm, as prefixes of increasing size from HubRankD's merit list are included into the hub set. We see that the dominating part of the running time is spent in iteratively computing missing PPVs. This part of the running time decreases steeply with increasing $|H|$, because large $|H|$ quickly limits the influence subgraph.

However, we can improve further beyond uniform power iteration over the influence subgraph, by concentrating on

```
1: inputs: edge conductances C, α, teleport r, tolerance ε_push,
   hubs H with precomputed PPVs
2: q ← r, p̂_r ← 0
3: while ‖q‖_1 > ε_push do
4:    pick node u with largest q(u) > 0 {delete-max}
5:    q̂ ← q(u), q(u) ← 0
6:    if u ∈ H then
7:       p̂_r ← p̂_r + q̂ PPV_u
8:    else
9:       p̂_r(u) ← p̂_r(u) + (1 − α)q̂
10:      for each out-neighbor v of u do
11:         q(v) ← q(v) + αC(v, u)q̂ {increase-key}
12: return personalized PageRank p̂_r
```

Fig. 15 Extending Proposition 4 to use hub PPVs

```
1: inputs: edge conductances C, α, teleport r, tolerance ε_push,
   hubs H with precomputed PPVs
2: q ← r, N_{H,r} ← 0, B_{H,r} ← 0
3: while ‖q‖_1 > ε_push do
4:    pick node u with largest q(u) > 0 {delete-max}
5:    q̂ ← q(u), q(u) ← 0
6:    if u ∈ H then
7:       B_{H,r}(u) ← B_{H,r}(u) + q̂
8:    else
9:       N_{H,r}(u) ← N_{H,r}(u) + (1 − α)q̂
10:      for each out-neighbor v of u do
11:         q(v) ← q(v) + αC(v, u)q̂ {increase-key}
12: return N_{H,r} + Σ_{h∈H} B_{H,r}(h) PPV_h
```

Fig. 16 More efficient version of Fig. 15

high-conductance paths and adapting the BCA algorithm to use judiciously chosen hubs. This is the subject of the rest of the paper.

4 HubRankP

Compared to whole-graph PageRank, HubRankD is a practical and efficient query-processing strategy. However, it has a few shortcomings.

- There are no theoretical guarantees about the accuracy of node scores or ranks when compared with the "gold standard" global PageRank.
- There is no direct control on the size of the influence subgraph. The total size of all PPVs over the influence subgraph may exceed main memory limits. (In our experiments, this happens in 2 out of 10,000 queries.)
- In case of a minor change in r from one query to another (say, a query word added or dropped), the influence subgraph needs to be recomputed from scratch. It can change dramatically, and there is no satisfactory way to reuse computations invested in earlier influence subgraphs.

These shortcomings are removed in HubRankP, described in this section. HubRankP is based on Berkhins's BCA described in Sect. 2.3.

4.1 Asynchronous PageRank with hubs

Suppose, for a selection of hub nodes $h \in H \subset V$, we have precomputed and stored $\text{PPV}_h \stackrel{\Delta}{=} p_{\delta_h}$. Then the pseudocode in Proposition 4 can be extended to use the hub PPVs, as shown in Fig. 15.

Proposition 7 *In Fig. 15, $\hat{p}_r + p_q$ is invariant across the while loop.*

Proof Let $q^<, \rho^<$ and $q^>, \rho^>$ be the values of q and $\hat{p}_r$ before and after one loop execution. First, consider the case $h \notin H$. Then

$$q^> = q^< - q^<(u)\delta_u + \alpha C q^<(u)\delta_u$$
$$\rho^> = \rho^< + (1-\alpha)q^<(u)\delta_u$$

We have to show that $\rho^< + p_{q^<} = \rho^> + p_{q^>}$. The rhs

$$\begin{aligned}
&= \rho^< + (1-\alpha)q^<(u)\delta_u + p_{q^<-q^<(u)\delta_u+\alpha C q^<(u)\delta_u} \\
&= \underline{\rho^<} + (1-\alpha)q^<(u)\delta_u \\
&\quad + \underline{p_{q^<}} - p_{q^<(u)\delta_u} + p_{\alpha C q^<(u)\delta_u} \quad \text{using (3)} \\
&= \underline{\text{lhs}} + (1-\alpha)q^<(u)\delta_u - p_{q^<(u)\delta_u} + p_{\alpha C q^<(u)\delta_u} \\
&= \text{lhs} + \mathbf{0}, \quad \text{using Theorem 1.}
\end{aligned}$$

Now, consider the case $h \in H$. In this case,

$$\begin{aligned}
q^> &= q^< - q^<(u)\delta_u \\
\rho^> &= \rho^< + q^<(u)p_{\delta_u}, \quad \text{from which} \\
\rho^> + p_{q^>} &= \rho^< + q^<(u)p_{\delta_u} + p_{q^<-q^<(u)\delta_u} \\
&= \rho^< + p_{q^<} + q^<(u)p_{\delta_u} - \underline{p_{q^<(u)\delta_u}} \quad \text{using (3)} \\
&= \text{lhs} + q^<(u)p_{\delta_u} - \underline{q^<(u)p_{\delta_u}} = \text{lhs}.
\end{aligned}$$

□

H has critical effect on query time: $u \in H$ *blocks* the expansion, while $u \notin H$ is *non-blocking* and propagates weight to out-neighbors v.

Step 7 in Fig. 15 accesses PPV_u from disk inside the loop. This can be avoided by adding up the coefficients $\hat{q}$ over the execution of the algorithm into an accumulator $B(u)$, which is multiplied by PPV_u probed from disk once, at the very end. Because B is a function of the hub set H and query teleport r, we will denote it as $B_{H,r}$. This version is shown in Fig. 16.

Proposition 8 *In Fig. 16,*

$$p_q + N_{H,r} + \sum_{h \in H} B_{H,r}(h)\,\text{PPV}_h$$

is invariant across the while loop.

The proof is very similar to that of Proposition 7 and is omitted.

In Fig. 15, we begin with $\hat{p}_r = \mathbf{0}$ and $q = r$, so $\hat{p}_r + p_q = p_r$. When we terminate, $q \approx \mathbf{0}$, so $p_q \approx \mathbf{0}$ and therefore $\hat{p}_r \approx p_r$. Similarly, in Fig. 16, $N_{H,r} + \sum_{h \in H} B_{H,r}(h)\,\mathrm{PPV}_h \approx p_r$ at termination.

Proposition 9 *At most* $\|r\|_1/(1-\alpha)\epsilon_{push}$ *pushes are executed before termination.*

Proof For the loop body to execute, $\|q\|_1 > \epsilon_{\text{push}}$. Therefore, $\|q\|_1$ decreases by at least ϵ_{push}, then, if $u \notin H$, it can increase again (if there is a self-loop, say) by at most $\alpha\epsilon_{\text{push}}$, because $\sum_v C(v,u) = 1$. Therefore, each execution of the loop body reduces $\|q\|_1$ by at least $(1-\alpha)\epsilon_{\text{push}}$. Because $q = r$ initially, the number of loop executions is at most $\|r\|_1/(1-\alpha)\epsilon_{\text{push}}$. □

In our experiments, we have indeed observed that the query execution time depends directly on the initial $\|r\|_1$ (details in Subsect. 4.2.4). We use a max-heap for q, so that we can drive the largest $q(u)$ down first. Note the parameter ϵ_{push} and the test $\|q\|_1 > \epsilon_{\text{push}}$ that trigger termination.

Residual data structure To terminate fast, we must "drain" q quickly, which is why we choose the u with the largest $q(u)$ to drain. (Correctness is guaranteed no matter which u we pick in step 4 in Fig. 16.) We implement q as a heap. A Fibonacci heap achieves $O(1)$-time insert and increase-key and $O(\log|V|)$-time delete and delete-max operations. Therefore, we used Chazelle's soft heap [16]. A soft heap costs $O(1)$ for *all* operations, but up to some small fixed fraction of keys in the heap may get arbitrarily corrupted. This only upsets the delete-max order; we can always retain the correct residual value, so that score correctness remains guaranteed. In practice, the soft heap reduces $\|q\|_1$ as quickly as the Fibonacci heap, and queries run twice as fast.

Very small footprint per query We can implement step 12 of Fig. 16 to minimize HubRankP's memory footprint, in a way not mentioned by Berkhin [8]. Because $N_{H,r}(u)$, $B_{H,r}(u)$, $\mathrm{PPV}_h(u) \geq 0\,\forall u$, we can return the top-$k$ nodes u with the largest $p_r(u)$ by indexing PPV_h in decreasing order of $\mathrm{PPV}_h(u)$ and pulling items from a cursor on the PPV_h record, sorting $N_{H,r}(u)$ and $B_{H,r}(u)$ likewise, and merging [21] until we output k hits. Given k is very small (10–20), this means the memory footprint of a query is essentially the number of nonzero elements in $N_{H,r}$ and $B_{H,r}$ in practice. This stands in stark contrast to HubRankD, where PPVs for each node in the active subgraph need to be materialized in RAM. We will see the benefits in Sect. 4.5.

4.2 BCA performance model

From the preceding description, we see that most of the work in updating $p_{r@(-1)}$ to $p_{r@0}$ is in computing $p_{r[M_0]}$. We

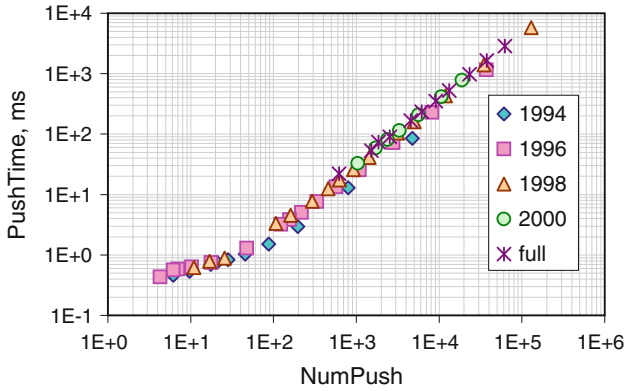

Fig. 17 Across all data slices, push time is proportional to the number of pushes, with a small cache effect for very small graphs

therefore want to select a hub set H that will reduce the average computation time of $p_{r[M_0]}$ as much as possible. To do this, we first need

- A distribution over match sets M (Subsect. 2.4.5).
- A predictive model for the execution time for $p_{r[M]}$ given M and H (Subsects. 4.2.1, 4.2.2, 4.2.3 and 4.2.4).

In this section, we will describe a predictive model for $\mathrm{PushTime}(H, \delta_u, \epsilon_{\text{push}})$. From Fig. 17, we see that PushTime is linearly proportional to the number of pushes, but it remains a challenging task to predict the number of pushes. Because of ϵ_{push} and hub set H, the push algorithm touches only a small portion of G, called the *active subgraph*, and denoted $\mathrm{PushActive}(H, \delta_u, \epsilon_{\text{push}})$. We establish (experimentally) that the number of pushes can be predicted well if we knew $\mathrm{PushActive}(H, \delta_u, \epsilon_{\text{push}})$, which in turn is impossible to predict without running the push algorithm itself. Therefore, we will proceed as follows:

1. We will show empirical evidence that the number of nodes in the active subgraph is an excellent predictor of total push time while executing a query.
2. Since the active subgraph is itself determined by the push algorithm, the above observation is not directly useful. We will provide an approximation to active subgraph size that also gives excellent predictions.
3. Even the approximation takes too long to compute naively on large graphs. We present a random-sampling algorithm to get an unbiased estimate of the approximation.

We will make sure we do not lose the essence of our goal quantity, push time, through these approximations. Despite our approximations, our hub selection strategy (Sect. 4.3) performs very well.

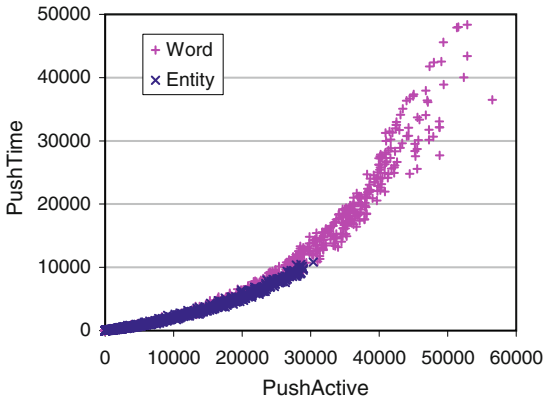

Fig. 18 PushActive$(H, \delta_u, \epsilon_{\text{push}})$ can predict push time (ms) accurately via regression

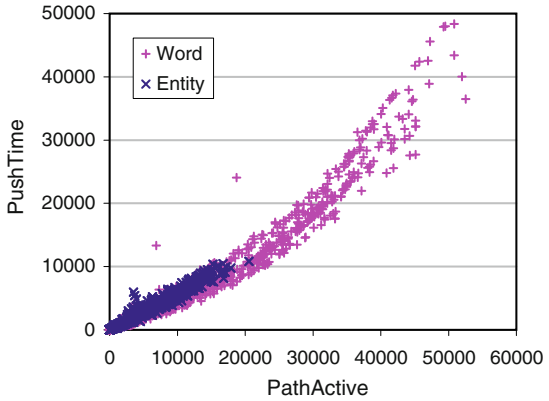

Fig. 19 Instead of PushActive$(H, \delta_u, \epsilon_{\text{push}})$, we can use PathActive$(H, \delta_u, \epsilon_{\text{push}})$ to predict push time (ms)

4.2.1 PushActive$(H, \delta_u, \epsilon_{push})$

If the active subgraph were acyclic, the push algorithm would insert each active node exactly once into the residual heap, and would complete in time proportional to PushActive$(H, \delta_u, \epsilon_{\text{push}})$. In general there would be loops and a node may enter and leave the residual heap q multiple times via increase-key and delete-max (see Fig. 16), so we expect the running time of the push algorithm to scale superlinearly with the number of nodes in PushActive$(H, \delta_u, \epsilon_{\text{push}})$. Figure 18 shows that the size of the active subgraph is a surprisingly accurate predictor of the total time spent by the push algorithm. As anticipated, the dependency is superlinear. Both word and entity origins are shown.

4.2.2 PathActive$(H, \delta_u, \epsilon_{push})$

Relating push time to PushActive$(H, \delta_u, \epsilon_{\text{push}})$ gives us insight into BCA's performance dynamics, but is not directly useful, because it is itself defined by a push run! As our second approximation step, we will consider only the single largest conductance path from u to each node, not touching any $h \in H$. The conductance of a path is the product of the conductance of its edges. Nodes not reached or nodes to which the path conductance is less than ϵ_{push} will be considered as not active.

We denote the shortest path tree (rooted at u) thus collected by PathActive$(H, \delta_u, \epsilon_{\text{push}})$, which is our approximation to PushActive$(H, \delta_u, \epsilon_{\text{push}})$. If the true active subgraph is already a tree, then no damage has been done; otherwise, PathActive$(H, \delta_u, \epsilon_{\text{push}})$ will generally be smaller. Figure 19 shows that the approximation is very acceptable.

4.2.3 Sampled estimate CohenActive$(H, \delta_u, \epsilon_{push})$

The hub selection algorithm in Sect. 4.3 will need estimates of PathActive$(H, \delta_u, \epsilon_{\text{push}})$ for a large batch of origin nodes u. Doing so many shortest path expansions explicitly and separately will be too expensive: we cannot afford time quadratic or worse in the size of the graph. We adapt an elegant random-sampling technique by Cohen [17] for quickly estimating the number of nodes reachable from each of a large number of origin nodes, within a given path length.

Consider a directed graph where edges have associated lengths (we will describe how to assign edge lengths based on the conductance matrix C shortly). Cohen's algorithm depends on the insight that, if we assign random scores to the nodes in the graph then the number of nodes reachable from node u is strongly related to the minimum score which can be reached from u. The confidence and accuracy of the estimate can be increased by repeating the estimation process N times, with different sets of random scores assigned to the nodes.

In iteration $1 \le i \le N$, we assign score $R_i(u) \in [0, 1]$ independently at random to each node $u \in V$. Let $g_i(u, d)$ be the node with minimum score that can be reached from u within a path length of d. Cohen estimated the number of nodes that can be reached from u within path length d as

$$\hat{s}(u, d) = \frac{N}{\sum_{1 \le i \le N} R_i(g_i(u, d))} - 1 \tag{6}$$

Let $S(u, d)$ be the nodes reachable from u within path length d, and let $s(u, d) = |S(u, d)|$. Cohen showed that for any $\epsilon > 0$ and for $u \in V$,

$$\Pr\left(|s(u, d) - \hat{s}(u, d)| \ge \epsilon s(u, d)\right) \le O\left(\tfrac{1}{\epsilon^2 N}\right).$$

Cohen also showed that, for all $u \in V$, the expected error

$$E\left(\frac{|s(u, d) - \hat{s}(u, d)|}{s(u, d)}\right) = O\left(1/\sqrt{N}\right).$$

In each of the N rounds (N is typically a small constant like 10), Cohen's algorithm first assigns random scores to nodes. This induces a total order on the nodes; we can regard the

```
1: inputs: G = (V, E), hubs H, conductances C
2: assign random scores to nodes, thereby inducing a total order
   v ∈ V = {1, 2, ..., |V|}
3: for each v ∈ V do
4:    initialize L(v) ← ( ), d_v ← ∞
5: for v ∈ V do
6:    initialize empty min-heap M
7:    insert v into M with key 0
8:    while M ≠ ∅ do
9:       remove (u, a) from M with smallest a
10:      insert (a, v) into L(u)
11:      d_u ← a
12:      if u ∈ H then
13:         continue
14:      for each in-neighbor w : (w, u) ∈ E do
15:         if w ∈ H then
16:            continue
17:         if (w, b) is in M then
18:            decrease key b to min{b, a − log C(u, w)}
19:         else if a − log C(j, k) < d_w then
20:            insert w into M with key d − log C(u, w)
21: return L(u) for all nodes u
```

Fig. 20 Preparing $L(u)$ at all nodes u to estimate CohenActive(H, δ_u, ϵ_{push})

node IDs as $\{1, 2, \ldots, |V|\}$. Then it builds a list $L(v)$ associated with each node v. The list contains pairs of the form $(a_v(i), u_v(i))$ in decreasing order of $a_v(i)$, such that for all i, $u_v(i) \in V$ is the node with the smallest ID among $S(v, a)$ for all $a_v(i) \leq a < a_v(i+1)$. It is easy to see that $u_v(i)$ are in increasing order, and, given query (v, a), $g(v, a)$ can be computed by binary-searching $L(v)$. Thus, we can avoid computing $s(v, a)$ for all v and all a in advance.

To apply Cohen's algorithm to our setting, we need to set the length of edge (u, v) to $-\log C(v, u)$. Figure 20 shows how to build $L(v)$ at all $v \in V$, for one round i of random score assignment. It is very important to update the definition of $g_i(u, d)$ to exclude nodes that are not reachable from u without passing through some $h \in H$. This results in extra checks in steps 12 and 15.

Following Cohen, we can show that the expected length of $L(v)$ is $O(\log |V|)$, and the running time of the algorithm shown in Fig. 20 is $O\big(|E| \log |V| + |V| \log^2 |V|\big)$ for one round. Probing $L(v)$ to compute $g(v, d; H)$ takes $O(\log \log |V|)$ time in expectation. With N rounds, the preprocessing time is $O\big(N|E| \log |V| + N|V| \log^2 |V|\big)$ and the expected probe time over lists $L_1(v), \ldots, L_N(v)$ is $O(N \log \log |V|)$. Each entry (v, d) in $L_i(u)$, where d is a distance, specifies that in the ith round, the node with minimum score node that can be reached from u without touching H and within distance d is v. We can find the score of v using $R_i(\cdot)$ and then obtain an estimate CohenActive(H, δ_u, ϵ_{push}) as $\hat{s}(u, -\log \epsilon_{\text{push}})$, using (6) (i.e., the path length cutoff d is $-\log \epsilon_{\text{push}}$ for us).

In our experiments, we used $N = 10$ as the best compromise between variability of random estimates and sampling speed. Final system performance is insensitive to fine-tuning N. We see from Fig. 21 that CohenActive(H, δ_u, ϵ_{push}) is a very usable surrogate for PathActive(H, δ_u, ϵ_{push}).

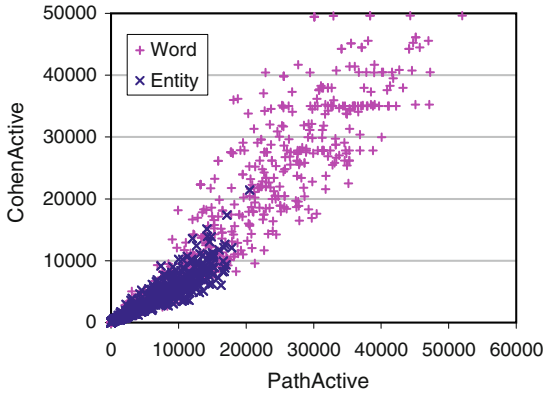

Fig. 21 CohenActive(H, δ_u, ϵ_{push}) estimates PathActive(H, δ_u, ϵ_{push}) fairly well

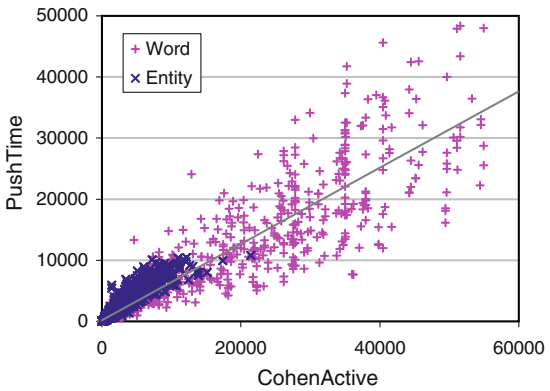

Fig. 22 End-to-end evaluation: From CohenActive(H, δ_u, ϵ_{push}), we can get a reasonable estimate of actual push time (in ms)

Finally, Fig. 22 shows that push time can be obtained via a regression from CohenActive(H, δ_u, ϵ_{push}). We call this the REGRESS($\cdot$) function. A linear fit can predict most push times to within a $2\times$ factor. The variance in Figs. 21 and 22 can be reduced by increasing N, but the hub set selection algorithm in Sect. 4.3 is tolerant to reasonable variance, so we can economize on preprocessing time.

4.2.4 Linearity and saturation

We have now completed a cost model for impulse teleport into one origin node. In Sect. 4.3, we will need to extend the cost model to general queries. A key question while doing so is whether CohenActive(H, r, ϵ_{push}) is reasonably *linear* in r. Figure 23 considers several origins u and plots the size of the active subgraph as the origin residual δ_u is scaled with a constant factor $0 < a < 1$.

The plots are not globally linear with respect to a; they show a steep, roughly linear growth followed by saturation to a fixed value. As a increases, push reaches more nodes until

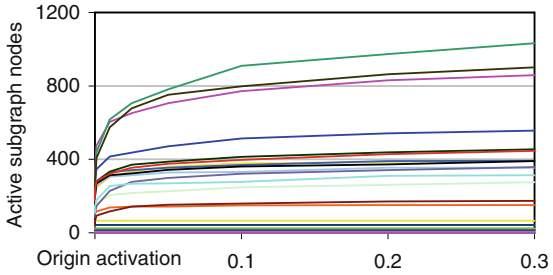

Fig. 23 Dependence of PathActive$(H, a\delta_u, \epsilon_{\text{push}})$ on $a > 0$

all nodes $R(\bar{H}, u)$ that are reachable from u without touching H (and within distance $-\log \epsilon_{\text{push}}$) are included in the active subgraph. This is consistent with *hop-plots* in social networks [23].

A key reason why PathActive is a good surrogate for PushActive is that, thanks to edge conductances being less than $\alpha < 1$, residuals in loops die down quickly. Ignoring these loopback edges, the aggregate push cost starting from origin u may be modeled as approximately additive over the push costs from the out-neighbors of u. Every edge traversed reduces the residual. Therefore, for a vast majority of nodes, we are in the small-a linear regime of Fig. 23. Summarizing, the following *linearity* and *additivity* properties hold reasonably well at most nodes during a query:

$$\text{CohenActive}(H, ar) \approx a\ \text{CohenActive}(H, r) \tag{7}$$

$$\text{CohenActive}(H, r_1 + r_2) \approx \text{CohenActive}(H, r_1) + \text{CohenActive}(H, r_2) \tag{8}$$

4.3 Hub set selection algorithm

We now use our understanding of the dynamics of the push algorithm to design approaches to select a good hub set H. Before we set out, we note that we can formulate this problem in several ways that are NP-hard, but the real utility lies in making the best approximations and proposing practical heuristics. We will do a cost-benefit style [25] greedy optimization.

Benefit: The marginal benefit of including a node u into H is the work saved because step 7 of Fig. 16 will terminate in node u without further weight propagation.

Cost: The cost is the space needed to store PPV_u. In this section, we will ignore the cost to compute PPV_u, but we will optimize and measure it in Sect. 5.

4.3.1 Work saved for one query if $u \in H$

If u is found in H in step 7 of Fig. 16, the residual $\hat{q}$ is "grounded" (i.e. not propagated to neighbors v), possibly saving a lot of pushes downstream. Consider a fixed query specified by teleport r. While executing the query, u is removed from the heap q a number of times. Let the sequence of these removal instances be $\text{DelMax}(r, u)$. For every delete operation $\rho \in \text{DelMax}(r, u)$, u has a residual score $\hat{q} = q(\rho, u)$.

Using additivity (8), we approximate the benefit of including node u in H for query r by estimating the times to compute some synthetic queries:

$$\text{WorkSaved}(H, r, u) = \sum_{\rho \in \text{DelMax}(r,u)} \text{PushTime}(H, q(\rho, u)\delta_u, \epsilon_{\text{push}})$$

(Note $q(\rho, u)$ is a scalar and δ_u a vector.) Using Sect. 4.2, the rhs is replaced by the regressed push time with u as the origin and $q(\rho, u)$ as initial residual:

$$\sum_{\rho \in \text{DelMax}(r,u)} \text{REGRESS}\left(\text{CohenActive}(H, q(\rho, u)\delta_u, \epsilon_{\text{push}})\right)$$

We will drop H and ϵ_{push} when they are fixed and clear from context. Assuming $\text{REGRESS}(\cdot)$ is roughly linear (least-squares fit in Fig. 22), we further approximate the work saved as proportional to:

$$\text{REGRESS}\left[\sum_{\rho \in \text{DelMax}(r,u)} \text{CohenActive}(q(\rho, u)\delta_u)\right]$$

This is still not directly useful, because, as in Sect. 4.2, we will not know $\text{DelMax}(r, u)$ without actually running the push algorithm. To get around this, we use the additivity property (8) because $q(\rho, u)$ is almost always tiny $\left(10^{-11} \ldots 10^{-5}\right)$ in practice:

$$\text{REGRESS}\left[\text{CohenActive}\left(\delta_u \sum_{\rho \in \text{DelMax}(r,u)} q(\rho, u)\right)\right]$$

Because u is not yet in H, step 9 of the algorithm in Fig. 16 immediately gives us $(1-\alpha)\sum_{\rho \in \text{DelMax}(r,u)} q(\rho, u) = N_{H,r}(u)$, and therefore the work saved for query r is

$$\text{REGRESS}\left[\text{CohenActive}\left(\frac{\delta_u N_{H,r}(u)}{1-\alpha}\right)\right],$$

which is $\text{PushTime}\left(H, \frac{\delta_u N_{H,r}(u)}{1-\alpha}, \epsilon_{\text{push}}\right)$.

4.3.2 Work saved by u over query workload

The work saved by indexing PPV_u, averaged over the query workload distribution $\tilde{f}(w)$ is

$$\sum_w \tilde{f}(w)\, \text{WorkSaved}(H, \delta_w, u)$$

By the previous discussion, we can rewrite this as

$$\sum_w \tilde{f}(w)\, \text{PushTime}\left(H, \frac{\delta_u N_{H,\delta_w}(u)}{1-\alpha}, \epsilon_{\text{push}}\right). \tag{9}$$

Given infinite CPU power, this can be estimated directly, but this involves computing $N_{H,\delta_w}(u)$ for *each pair* w, u. The solution is to use the approximate linearity property (7) because $N_{H,\delta_w}(u)$ is very small for sufficiently large graphs, thus letting us approximate

$$(9) \approx \sum_w \tilde{f}(w) \frac{N_{H,\delta_w}(u)}{1-\alpha} \text{PushTime}(H, \delta_u, \epsilon_{\text{push}})$$
$$= \frac{\text{PushTime}(H, \delta_u, \epsilon_{\text{push}})}{1-\alpha} \sum_w \tilde{f}(w) N_{H,\delta_w}(u),$$

the advantage being that $\sum_w \tilde{f}(w) N_{H,\delta_w}(u) = N_{H,\tilde{f}}(u)$, so a single PageRank computation with $r = \tilde{f}$ suffices. Also recall that $\text{PushTime}(H, \delta_u, \epsilon_{\text{push}})$ can be quickly estimated for all u together using Subsect. 4.2.3.
Summarizing, our final expression is

$$(9) \approx \frac{1}{1-\alpha} \text{PushTime}(H, \delta_u, \epsilon_{\text{push}}) N_{H,\tilde{f}}(u)$$

Let us sanity-check the final expression. If u is in the backwaters of G, not reaching many nodes, or if H already blocks many paths out of u, then pushes starting at u end quickly anyway, so indexing PPV_u is not very profitable. This shows up in $\text{PushTime}(H, \delta_u, \epsilon_{\text{push}})$. If a lot of frequent words reach u often, it may be profitable to index PPV_u. This is reflected in $N_{H,\tilde{f}}(u)$.

4.3.3 PPV_u *storage space*

Having modeled the benefits, let us model the cost of PPV_u: the space taken by PPV_u on disk. Let $R(u)$ be the nodes reachable from u. Assuming no node v has vanishingly small $\text{PPV}_u(v)$ score, PPV_u stored to full precision may need up to $|R(u)|$ node IDs and their $|R(u)|$ corresponding scores, which would be prohibitive for even modest H. Aggressive clipping (removing PPV elements below a threshold ϵ_{clip}) hardly damages ranking accuracy while reducing index space, as shown in Fig. 24 (and also used by Balmin et al.[6]). Our problem is that our cost-benefit hub selector will need space estimates for the *clipped* PPVs.

As in the estimation of work saved, this is also a difficult problem that we can address only in aggregate statistics; it is not possible to compute the number of post-clip elements in a PPV perfectly without computing the PPV first!

The solution is to use the property that the elements of PPV_u tend to be power-law distributed in social networks [44]; if they are sorted in decreasing order, $\text{PPV}_u(v_i) \propto i^{-\beta}$, approximately. One key question is whether one universal power β will suffice reasonably for all PPVs in a fixed graph G.

We rescale all PPVs so that $\text{PPV}_u(v_1) = 1$, let x_i be the log of the rank of v_i (i.e., $\log i$), $y_i = \log \text{PPV}_u(v_i)$, and fit a regression $y = -\beta x$ given (x_i, y_i) observations

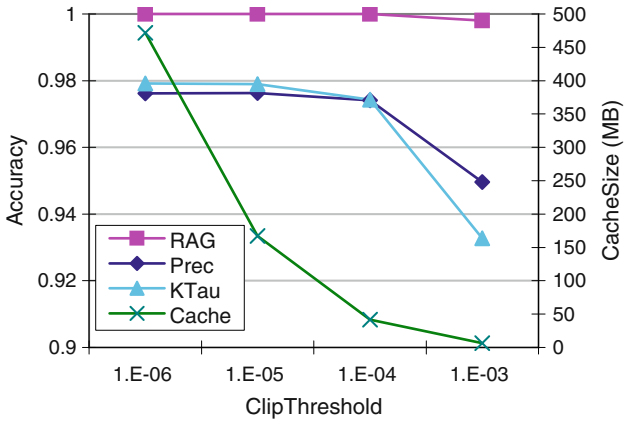

Fig. 24 Clipping PPVs have very modest effect on accuracy (see Subsect. 2.4.6 for accuracy definitions) while reducing PPV cache size drastically

(collected over many PPVs). A least-square fit yields $\beta^* = -\sum_i x_i y_i \,/ \sum_i x_i^2$. To get an impression of the stability of β^*, we took 50 random samples each of 5% of word and entities as origins, computed their PPVs, and used those for a least-square β fit; clearly, a single β^* for a given graph is not a bad choice:

Quantity	CiteSeer1994	CiteSeer1998
Minimum β^*	1.36	1.75
Maximum β^*	1.39	1.81
Average β^*	1.37	1.77
Std dev of β^*	0.083	0.015

Therefore, it seems reasonable to trust a single power law for all PPVs of a given graph. Perhaps surprisingly, $|R(u)|$, β^*, and PPV clip threshold ϵ_{clip} directly determine the clipped PPV size estimate:

1: **input:** unclipped PPV size estimate $|R(u)|$, clip threshold ϵ_{clip}, fitted power β^*
2: $unscaledSum \leftarrow \sum_{i=1}^{|R(u)|} i^{-\beta^*}$
3: $rawClippedSize \leftarrow \min\left\{i : \frac{i^{-\beta^*}}{unscaledSum} < \epsilon_{\text{clip}}\right\}$
4: **return** $\min\{rawClippedSize, |R(u)|\}$

To complete the job, we need $|R(u)|$ for all origins u. We can again use Cohen's sampling algorithm with special input parameters: $\text{CohenActive}(H = \varnothing, \delta_u, \epsilon_{\text{push}} = 0)$ returns an estimate of $R(u)$, the number of nodes reachable from origin u. Results are shown in Fig. 25; they are adequately accurate for good hub selection.

4.3.4 *Hub inclusion policies*

LPR The baseline "large PageRank" or **LPR** policy suggested by Jeh and Widom [32] and Berkhin [8] orders entity nodes in decreasing global PageRank order with uniform teleport $r = \mathbf{1}/|V|$.

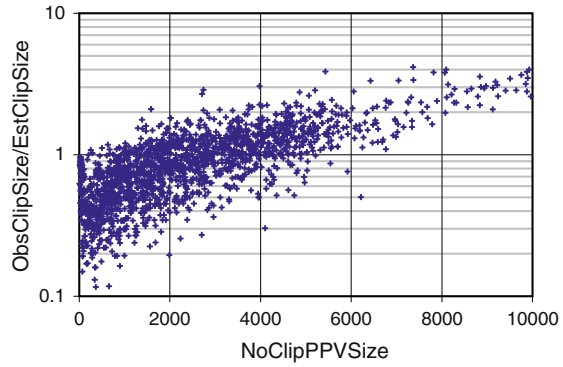

Fig. 25 Over a wide range of unclipped sizes, the ratio of observed to estimated clipped sizes are within a factor of 2 for the vast majority of PPV origins

1: **inputs:** target $|H|$, PPV clip threshold ϵ_{clip}, batchSize
2: find CohenActive$(\varnothing, \delta_u, \epsilon_{\text{push}} = 0)$ for all $u \in V$
3: thus estimate clipped sizes of all PPV_u {cost}
4: $H \leftarrow \varnothing$
5: **while** H is not large enough **do**
6: for all nodes u not in H yet compute {benefit} $\sum_w \tilde{f}(w)\,\text{WorkSaved}(H, \delta_w, u)$
7: set $\text{merit}(u) = \text{benefit}(u)/\text{cost}(u)$
8: greedily include best batchSize nodes in H
9: **return** selected hub set H

Fig. 26 LAP hub selection

N1 HubRank [12] ordered hubs u one-shot by only $N_{H,\tilde{f}}(u)$, ignoring the important downstream work term $\text{PushTime}(H, \delta_u, \epsilon_{\text{push}})$ that "looks ahead" from u. We call this "naive one-shot" or **N1**.

LAP The cost/benefit method proposed here as part of HubRankP is called "lookahead progressive" or **LAP**. LAP balances query time benefit against PPV space cost and runs a greedy packing procedure shown in Fig. 26.

The batched updates to H account for reduction in merit of nodes owing to nodes included in earlier batches. The loop body is fairly expensive, so we should amortize it over a reasonably large batchSize. At the same time, excessively large batches will fail to recognize that earlier hub nodes render later candidates less useful. In practice, we have found the quality of H insensitive to a broad range of batchSize.

4.4 Handling streams of teleports

A key feature of HubRankP, unlike HubRankD, is that HubRankP can recompute node scores quickly as r changes incrementally. Query-time personalization of PageRank is increasingly unavoidable [4,6,14,18] in applications where user activity must be incorporated dynamically into the graph. In this section, we describe how HubRankP can be used for online personalization.

To design r at any instant, the system may draw on past profile information, and/or use previous queries and browsing action within the current session. As an example from Fig. 1, suppose, after an initial search, J decides that a reviewer from industry (as against academia) would be preferable. J adds the node representing the type "companies" to the set of match nodes. J may also browse some similar papers and consider some other reviewers along the way.

For simplicity, assume that the user does not switch task or goal and is in a single "session". Keywords and entities alike are nodes in G. At every time step t, the user indicates a (typically small) subset of nodes $M_t \subset V$ as matched nodes. M_t may include both words (query modification) and entities (click/browse actions).

The current time step is designated 0. We have already observed $M_{-T}, \ldots, M_{-1}, M_0$. We need to define the corresponding teleports and thereby the personalized PageRanks for each step. Specifically, given recent match sets and p_{last}, we want to quickly compute r_{now} and p_{now}.

The data stream literature [5,36] suggests several ways to design teleport vectors given a stream of recent match sets. HubRankP fully supports all of them, but evaluating which gives the most meaningful rankings is an important area for future work.

Before proceeding, we need one piece of notation. For a specific match node set M, a conventional teleport vector is $r[M]$, given by $r[M]_u = 1/M$ if $u \in M$ and 0 if $u \notin M$. Our system can work with a more general definition of $r[M]$.

4.4.1 Indefinite accumulation

We average the teleport over the whole session:

$$r_{\text{now}} = \frac{1}{T+1} \sum_{t=-T}^{0} r[M_t]$$

If we have seen match sets $M_{-T}, \ldots, M_{-1}$ and already computed p_{last}, and we now see M_0, we can update

$$p_{\text{now}} = \frac{T}{T+1} p_{\text{last}} + \frac{1}{T+1} p_{r[M_0]}$$

4.4.2 Finite window

We only remember the last W match node sets: assuming $T \geq W$,

$$r_{\text{now}} = \frac{1}{W} \sum_{t=-W+1}^{0} r[M_t]$$

Once we are in the steady state, $(T \geq W)$, we have to simply drop $p_{r[M_{-W}]}$ and add on $p_{r[M_0]}$:

$$p_{\text{now}} = p_{\text{last}} - \frac{1}{W} p_{r[M_{-W}]} + \frac{1}{W} p_{r[M_0]}$$

In a small-scale system (say, personalized desktop search), we might hang on to all $p_{r[M_t]}$ for t in the window, but for large-scale shared search that may consume too much RAM, and it may be cheaper to store only Ms and recompute $p_{r[M_{-W}]}$.

Exponential decay We remember old match node sets, but give them less importance than recent ones:

$$r_{\text{now}} = \sum_{t=-T+1}^{0} c^t r[M_t], \quad \text{where } c < 1.$$

In steady state, this is the infinite sum

$$r_{\text{now}} = \sum_{t \geq 0} c^{-t} r[M_{-t}].$$

This also has the effect of forgetting old match sets, but more smoothly. This is again a simple matter of scaling and adding:

$$p_{\text{now}} = c\, p_{\text{last}} + p_{r[M_0]}$$

In all cases, r_{now} keeps changing through the session, and p_{now} must be recomputed quickly with each change. Therefore, PageRank cannot be computed offline and used as a static score. On the other hand, query-time whole-graph PageRank computation is unacceptably slow.

Hanging on to recent PageRank vectors may consume much RAM. In practice, the user needs to see only some top-k items of p_{now}. We can keep around a safe overdose of top scores from p_{last} and merge [21] with $p_{r[M_0]}$. We expect that this would not introduce significant ranking errors over short sessions.

4.5 HubRankP experiments

We report the performance of HubRankP and compare with other systems where appropriate. Details of our testbed and evaluation measures are in Subsect. 2.4.

Hub inclusion policy comparison Fig. 27 shows that LAP is better than N1 and much better than LPR. This shows that LPR, without the benefit of HubRankP's model and algorithms, results in $> 10\times$ query times as LAP. As H grows, the more elaborate LAP selection beats the more crude N1 method. Note that part (a) uses $|H|$ as the x-axis, whereas part (b) uses the actual index size, which is more natural.

A preliminary implementation of HubRank [12] used the **N1** hub node selection strategy described here, but, instead of storing truncated PPVs, it stored Monte-Carlo approximations of PPVs called *fingerprints* [24]. We call this older version of HubRank **N1FP**. For completeness, we also report on a head-to-head comparison, using the exact same data and testbed, between HubRankP and **N1FP**. See upper right corner of Fig. 27. LAP with 30 MB of PPV index beats N1FP with 50 MB of FP index by almost three orders of magnitude in query speed.

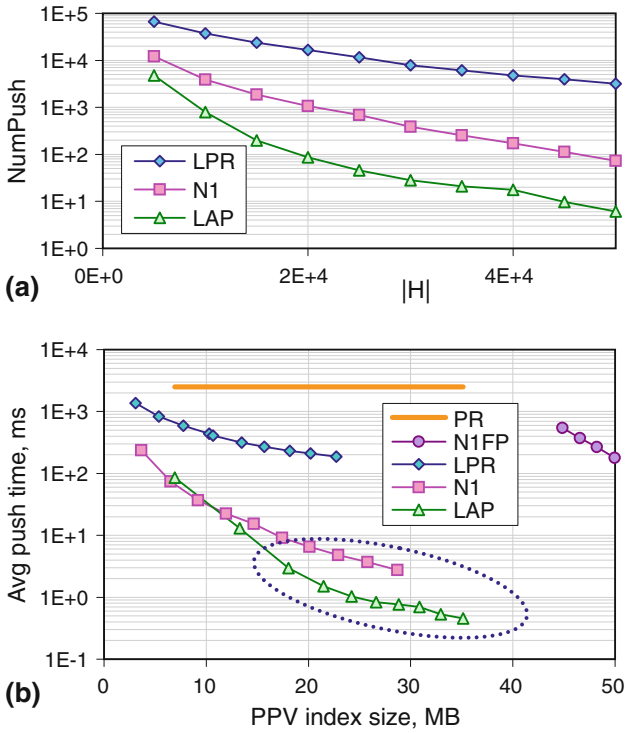

Fig. 27 Comparison of LAP, N1, LPR orders for the 1994 snapshot. Note: A text index needs 55 MB; y-axis is log-scale. Whole-graph power iteration (PR) xtime is over four orders of magnitude larger

In retrospect, truncated PPVs provide a better trade-off than fingerprints with respect to all performance metrics: index space, query-processing time, and even ranking accuracy: N1FP has typical RAG, precision and τ accuracy at rank 20 (Sect. 2.4.6 defines these) of **0.996, 0.916, and 0.829**, while HubRankP (LAP with PPV) achieves scores of **0.998, 0.95 and 0.94**.

Effect of scaling G and H together We used temporal snapshots of CiteSeer, as described in Subsect. 2.4.2, to evaluate the scalability of HubRankP. A critical issue is how quickly H needs to scale with G so that the query time remains independent of the size of G. $|H|/|V|$ is not a very accurate measure of index scaling requirements, because PPVs for nodes in H may have diverse sizes on disk after clipping at threshold ϵ_{clip}. A more faithful measure of relative storage overhead of indexing is to divide the total number of PPV elements (node ID and node score) across all PPVs by the number of edges in the ER graph. This is the quantity used for the x-axis in Fig. 28.

Note that a $8\times$ storage inflation (on disk) compared to the edge list is very modest in absolute terms, compared to the storage required for the text and text index (see the detailed comparison for 1994 in Table 4). Figure 28(a) shows that, for

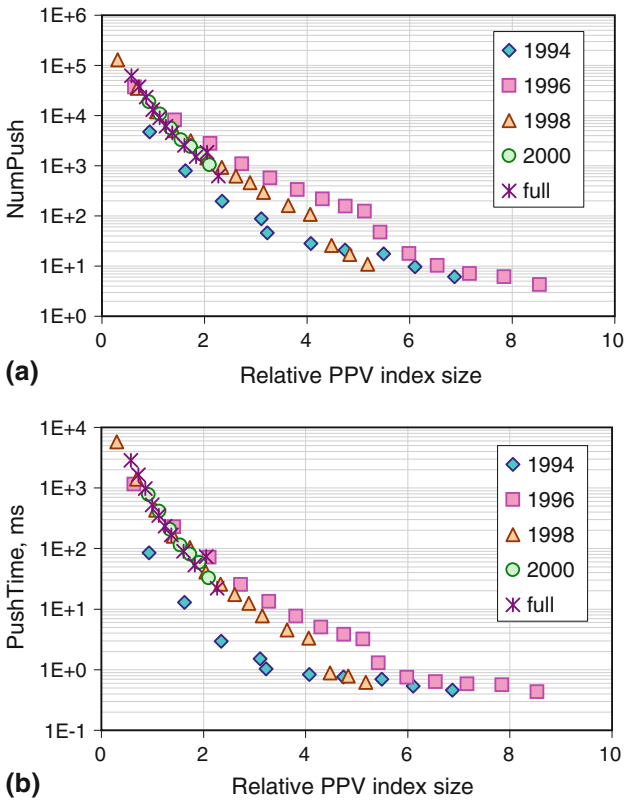

Fig. 28 By scaling H with G, **(a)** the number of pushes and **(b)** query time can be held essentially fixed. Each curve is for a distinct CITESEER snapshot year, shown in the legend

a fixed index overhead, HUBRANKP can keep the absolute number of pushes within one order of magnitude notwithstanding diverse graph sizes. Push times in Fig. 28(b) are also close, except for the smallest data set (1994) that gets some cache advantage. In fact, the largest data sets (2000 and full) fare *better* than the 1996 slice.

Summarizing, as long as H scales with G, query time can be kept constant independent of graph size. In contrast, global PageRank times scale up steeply with $|V|$ and $|E|$.

It might sound dangerous to allow H to scale with G because of index space and indexing time considerations. However, 1. the PPV index is on disk, 2. on clipping, the PPV space required reduces drastically (compare the x-axis of Fig. 27 with 55 MB, the Lucene index size used for this chart.), and 3. in Sect. 5.2 we show that the time to build the PPV index can be reduced substantially by judiciously exploiting PPVs computed earlier to compute later PPVs.

Query-time footprint As mentioned in Sect. 4.1, HUBRANKP needs RAM that is roughly the size of the active subgraph. Figure 29 shows that HUBRANKP query footprint (average 1119 node-score records) is over an order of magnitude smaller than HUBRANKD's (average 17102 node-score records). This makes HUBRANKP better suited to extending from personal desktop search to shared enterprise-scale search.

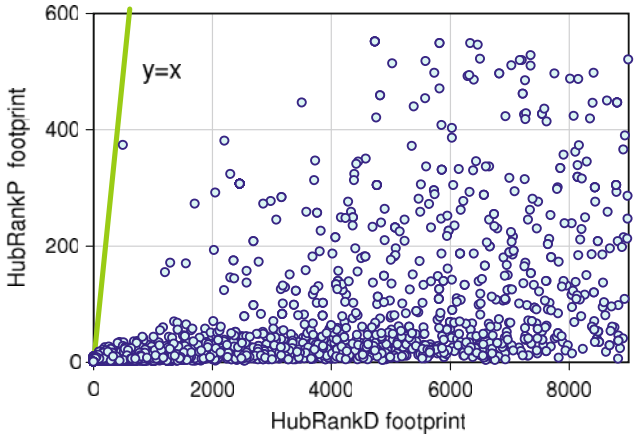

Fig. 29 HUBRANKP has much smaller RAM footprint per query than HUBRANKD. Note the different x and y scales

Dynamic teleport update performance We evaluate HUBRANKP using a realistic model of a live user searching and browsing an E-R graph database. A suitable user interface might have two panels. The left panel shows a ranked list of top-k responses. Clicking on a node displays it in the right panel, and updates the match node set M_t. This is then used to instantly recompute the personalized PageRank, and a new ranked list is presented in the left panel, while the user is viewing the right panel. Each session begins with a keyword query. For every browse step, the user picks a page among the top-k at random. For simplicity, all sessions involve a small fixed number of browse steps (here, 5).

The model can be made more realistic in a few ways: we can generate the session length randomly, and users may traverse relationship links to browse from entity to entity without referring to the left panel showing the ranked list. Still, we believe we capture the essence of real-time personalization in our simple proposal.

Lacking realistic click trail data from CITESEER, we continue to prepare the PPV index based only on keyword queries in CITESEER's query logs. In other words, the PPV index is optimized only for the initial keyword query and not the subsequent browsing steps. Figure 30 shows, for several values of $|H|$, the average time taken for the initial keyword query followed by the average time taken by the five dynamically personalized browsing steps.

Query time drops markedly from the keyword query to the first browsing step, then gently increases during subsequent hops. The initial drop suggests that the push algorithm finds blocking hub nodes during the first browse step even more readily than during the execution of the initial keyword query. This hints that LAP, like N1, is including a generous number of entity nodes as hubs. (This is confirmed in the

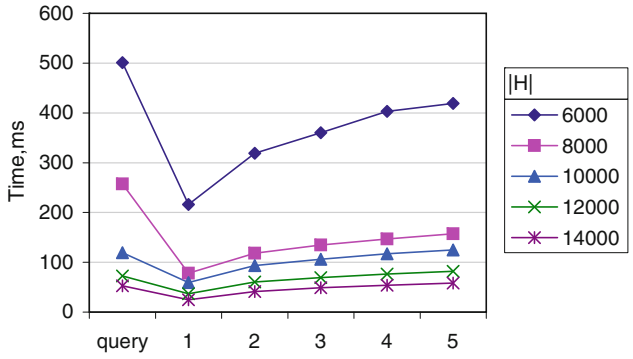

Fig. 30 As the user types an initial keyword query and browses away by up to 5 hops, we can personalize the ranking on the fly at speeds comparable to the initial query

Table 5 Both word and entity PPVs are essential for fast query processing

PPV Index (MB)	$\|H\|$	AvgQueryTime (ms)
Only entities in H		
7	17,307	3,758
8	27,465	1,124
10	32,687	771
11.3	37,879	443
Only words in H (like ObjectRank)		
7	9,308	9,122
8	11,028	5,880
10	14,806	5,443
11.3	17,405	5,187
Both words and entities in H (**HubRankP**)		
7.31	8,000	**748**
8.86	10,000	**288**
10.2	12,000	**132**
11.3	14,000	**72**

Bold figures show that using both word and entity hubs results in drastically reduced average query times

next experiment.) As we move further away from the initial query, the lack of optimized hubs shows through in gradually increasing push times. However, the time needed to update the ranked list typically remains comparable to the time taken by the initial keyword query.

Both entity and word hubs exploited LAP does an elaborate cost-benefit analysis to select both words and entities in H. Is such an elaborate scheme necessary, even for keyword query workloads? Could we, say, simply pick some words that are frequent in the query log and cache their PPVs, minimally extending ObjectRank? Alternatively, could we cache only entity PPVs, because word PPVs are usually larger in size and therefore are less value for index storage than entity PPVs?

To check this, we retained our hub merit order, but did separate runs with only word PPVs, only entity PPVs, and both. On omitting words or entities, some queries became so slow that global PageRank became faster than push, so we considered that as the query time.

Table 5 makes it clear that our hub selection really exploits synergy between word and entity selection. If only word PPVs are indexed and even one query word "misses" we need to invoke the push algorithm. With no entity blockers available, this can be slow. Entity-only is also visibly slower than word+entity. This also explains why, in Fig. 30, updates based on clicks on entity nodes were very fast despite training on word match sets alone.

4.6 Top-K ranking

The query-processing algorithm presented thus far is capable of computing personalized PageRank of all nodes up to some small additive error. However, for typical ad hoc search applications, scores for all items are not needed; the quest is for a few top-scoring items. In the context of personalized PageRank for ad hoc queries, the user is not really looking for the whole vector p_r as a solution to $p_r = \alpha C p_r + (1-\alpha)r$, but is looking for the K nodes that have the largest PageRank scores in p_r.

Traditionally, power iteration methods iterate the assignment $\tilde{p}_r^{(t+1)} \leftarrow \alpha C \tilde{p}_r^{(t)} + (1-\alpha)r$, until $\left\| \tilde{p}_r^{(t+1)} - \tilde{p}_r^{(t)} \right\|$ becomes "small", i.e., the scores stabilize. Strictly speaking, score stability is no guarantee of rank stability [37]. Besides, in the context of HubRankP, where there is no notion of a global iteration, and node scores change by tiny amounts over a large number of push steps, it is unclear how to test for even score stability.

However, a corollary of Propositions 7 and 8 is that for each node u, the estimated PageRank $\hat{p}_r(u)$ is always less than or equal to the true PageRank $p_r(u)$. Suppose we could also estimate, for each node u, a nontrivial *upper bound* $\check{p}_r(u)$. If, at any time during the execution of the push algorithm in Figs. 15 or 16, we can find a set of K nodes $u_1, \ldots, u_K$ such that for all $v \notin \{u_1, \ldots, u_K\}$, $\check{p}_r(v) < \min_{1 \le k \le K} \hat{p}_r(u_k)$, then we would at least know the identity of the top-K nodes and terminate the push loop. Such techniques have been used for top-K queries on relational data for a long time [21,50], but not for PageRank.

A small relaxation of the above termination condition often leads to even faster termination. When the user specifies that she wants to view the top K answers, it is usually quite acceptable if a few more answers are returned. For example, if the user wants $K = 20$ answers, the system may limit the number of answers to a range $\left[\underline{K}, \overline{K}\right]$, say, [20,40]. In this case, the termination check above should be modified to looking for a set of K nodes $\{u_1, \ldots, u_K\}$, where $\underline{K} \le K \le \overline{K}$, such that for all $v \notin \{u_1, \ldots, u_K\}$, $\check{p}_r(v) < \min_{1 \le k \le K} \hat{p}_r(u_k)$. The termination test will itself take some

time, so we should engage in it only after we have spent some substantial time in push steps. Subject to this condition, the bound estimation and termination check would be inserted at the end of the main push loop, i.e., just after line 11 in Fig. 15.

All that remains is to design the upper bound $\check{p}_r$. There are simple, loose bounds and there are more careful bounds. Computing the simple, loose bounds and checking for termination are so fast that they typically compensate for their looseness.

Proposition 10 *In the push algorithm shown in Fig. 15, at any time, for all nodes u,*

$$\hat{p}_r(u) \leq p_r(u) \leq \hat{p}_r(u) + \|q\|_1 \stackrel{\text{def}}{=} \check{p}_r(u).$$

This is a very loose and yet surprisingly effective bound: it just says that the entire residual q could land up at any single node. Note that, to compute $\check{p}_r$, we need to know $\hat{p}_r$, which means we cannot keep $N_{H,r}$ and $B_{H,r}$ separate as in Fig. 16, but must combine them, at least every time we do a termination check. However, the PPV database will typically use a cache, and there is high locality of PPV access in practice. That $\check{p}_r(u) - \hat{p}_r(u)$ is a constant independent of u (for a fixed current residual q) makes implementation very easy and fast.

Note that the top-K algorithm returns a set of top-K nodes, but the exact order among them is not known. Unlike in relational top-K query systems [21,50], we know of no straightforward way to update $\hat{p}_r(u)$ to $p_r(u)$ for all top-K finalists. In experiments, with aggressive values of ϵ_{push} (10^{-5} to 10^{-3}), if the finalists are ordered by $\hat{p}_r(u)$, ranking errors are exceedingly rare.

Figure 31 shows that relaxing K to $\left[\underline{K}, \overline{K}\right]$ leads to substantially increased success rate for termination check. In typical runs, about 40% of the queries terminate early. At the same time, the actual rank K^* at which the termination check succeeds is typically very close to $\underline{K}$.

As Fig. 32 shows, termination check is fast and effective. Tests for termination take a very small amount of time (which we limit, as discussed earlier) and typically give a considerable speed boost. Also reassuring is that as little as 4% time invested in quit checks result in robust gains. The overall query time is very insensitive to the fraction of time allocated to termination checking.

More sophisticated bounds are possible [27], but if we account for the complexity of computing them and implementing the corresponding termination checks, gives but meager speedups.

One final note is in order. The entire index and query optimization framework depends on the push time estimation of Sect. 4.2, which has no notion of early termination. Therefore, strictly speaking, if early termination is used, the performance model of Sect. 4.2 gives us only an upper bound

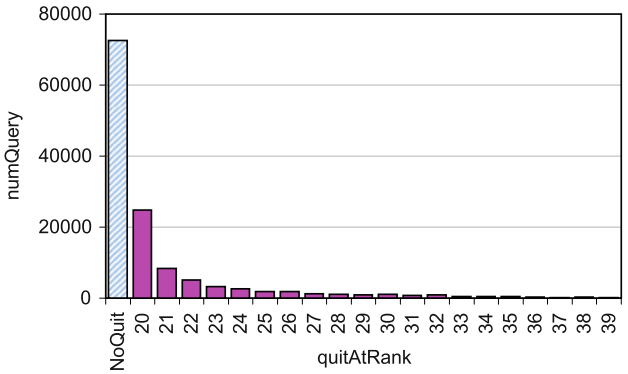

Fig. 31 Relaxing K to $\left[\underline{K}, \overline{K}\right]$ (= [20, 40] here) enables more frequent quits, while statistically the quit often happens close to $\underline{K}$. The leftmost bar corresponds to no early termination

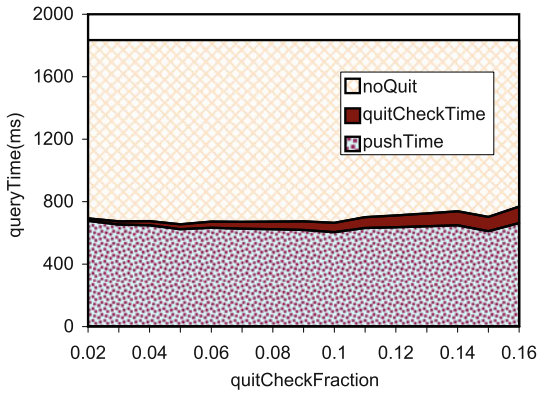

Fig. 32 Using Proposition 10, push times averaged across queries vs. fraction of push time allowed in termination checks. (The top line uses no termination checks.)

on query time. At this time, we see no way of modeling the (highly unpredictable) effects of early termination on push time. However,

- The typical multiplicative error of our push time estimates is of the same order as the gains of early termination, so the additional inaccuracy introduced into the model is not overwhelming.
- In any case, in the interest of tractability, we use a greedy heuristic for hub inclusion, not a global optimization with perfect inputs.
- In our experience, the estimation inaccuracy may lead to some local mistakes in hub merit list *ordering*, but if H is of reasonable size, the *set* of hub nodes selected is adequate.

5 Building the PPV index

We finally turn to the question of populating the hub index efficiently, once the hub set H has been decided upon. This is required for running both HubRankD and HubRankP.

1: pre-estimate fixed Power Iteration time t_{PI}
2: $H_{\mathrm{done}} = \varnothing$
3: **while** $H \neq \varnothing$ **do**
4: **batch:** $t_{\mathrm{push}}(h) \leftarrow \mathrm{PushTime}(H_{\mathrm{done}}, \delta_h) \forall h \in H$
5: **batch:** reorder nodes in H (Section 5.1)
6: remove next origin h from H and get $t_{\mathrm{push}}(h)$
7: **if** $t_{\mathrm{push}}(h) < t_{\mathrm{PI}}$ **then** compute PPV_h using push
8: **else** compute PPV_h using Power Iterations
9: include h in H_{done} and index PPV_h

Fig. 33 Populating the hub index

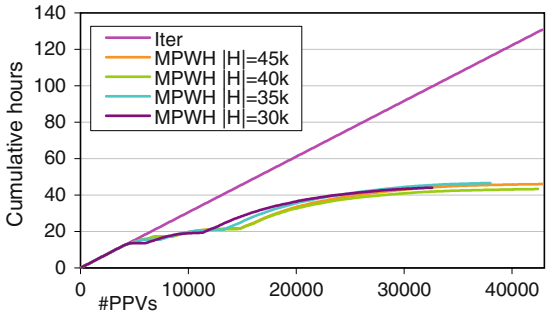

Fig. 34 While early PPVs use power iteration, later PPVs leverage them using BCA and run faster and faster

The baseline approach is to compute PPV_h for each $h \in H$ independently using power iterations, which, in practice, takes time $\propto |H|(|E| + |V|)$.

In this section, we propose a way to compute hub PPVs more efficiently. As some PPVs in H become available, computing another PPV_h is essentially a "query" with $r = \delta_h$. So the PPVs already available can speed up PPV computation.

In Fig. 33, "**batch:**" statements are executed once in a while to amortize over computational costs. PushTime estimation in Step 4 uses Sect. 4.2 extensively.

5.1 The MPWH hub ordering

The order in which hub PPVs are computed is important because PPVs computed earlier reduce subsequent work. Our modeling exercise in Subsect. 3.1.4 and Sect. 4.2 were already nontrivial. It seems exceedingly difficult to estimate the total time needed by the algorithm in Fig. 33 as a function of the ordering policy, leave alone design the optimal ordering policy. After evaluating several heuristic policies, we realized that we should first schedule nodes h that block many "heavy" paths from other (pending) hubs. To estimate this, we create a teleport vector $r[H]$ with $r[H](h) = 1/|H|$ if $h \in H$ and 0 if $h \notin H$, and then we compute the personalized PageRank (PPR) $p_{r[H]}$. The largest elements of $p_{r[H]}$ give us nodes to schedule next. Like push time, $p_{r[H]}(h)$ keeps changing because H is changing. To save costs, we update the hub order only occasionally. We call this *maximum PageRank wrt H* or MPWH ordering.

5.2 PPV indexing experiments

Figure 34 shows that once a critical mass is reached by H, more PPVs can be added to H much faster than power iterations. Each line is for a different *final* $|H|$ (but this hardly matters). While the early PPVs are computed using power iterations, later PPVs are computed much faster by exploiting PPVs already in H. The end effect is that the total PPV indexing time quickly levels out as $|H|$ is increased. This shows that scaling up H mildly with G is a practical proposition. We also note that power iteration itself can be sped up substantially by recent acceleration techniques [33].

6 Conclusion

Personalized PageRank, also known as random walk with restarts, is useful not only for personalization of Web search, but also as a general way of ranking nodes in E-R graphs in response to sparse but fast-changing teleports. Several algorithms and systems compress [2] massive link graphs into Connectivity Servers [10], but we know of no systems that provide indexing for fast, dynamic computation of personalized PageRank and related graph proximity measures.

We presented a generic framework called HubRank for preparing a PPV index, and exploiting it during query execution. Within this framework we considered two query execution strategies. HubRankD uses Jeh and Widom's [32] decomposition theorem as a query execution mechanism, whereas HubRankP uses Berkhin's [8] BCA. None of the earlier personalization proposals gave a workload-cognizant hub selection strategy, which is the focus of this paper. Our specific goal was to enable *constant time* personalized PageRank, irrespective of the size of the graph.

We discussed in detail a number of important engineering issues. We first gave predictive performance models for our query processors. Then we gave effective, workload-driven algorithms for selecting good hub sets. Finally, we showed how to precompute PPVs quickly.

Experiments with CiteSeer and US Patents entity-relationship graphs, together with millions of real queries from CiteSeer, showed that HubRank achieves small index space, index-building time and query-processing time while maintaining high ranking accuracy. While both approaches are of interest, our experience suggests that HubRankP is better suited to handling clickstreams as online personalization input.

Apart from personalized PageRank, hitting and commute times have been proposed as other viable notions of graph proximity by Sarkar et al. [47]. Unlike our deterministic bounding techniques in Sect. 4.6, their algorithm uses a more aggressive probabilistic pruning, with approximation guarantees on the resulting ranking. They do not use query log

statistics or any caching or indexing technique. Their reported query times are about an order of magnitude larger on the same data set on comparable hardware. It would be of interest to extend our PPV indexing framework to also support and speed up hitting and commute time queries. That would take us closer to a generic graph database with indexing and query optimization support for graph proximity search.

Acknowledgments Thanks to C. Lee Giles for CITESEER data, to Yannis Papakonstantinou for access to OBJECTRANK source code, and Pavel Berkhin and Andrei Broder for helpful discussions. Thanks to Ganesh Ramakrishnan, Somnath Banerjee and Devshree Sane for proofreading.

References

1. Abiteboul, S., Preda, M., Cobena, G.: Adaptive on-line page importance computation. In: WWW Conference, pp. 280–290 (2003)
2. Adler, M., Mitzenmacher, M.: Towards compressing Web graphs. In: Data Compression Conference, pp. 203–212 (2001)
3. Agrawal, S., Chaudhuri, S., Das, G.: DBXplorer: A system for keyword-based search over relational databases. In: ICDE. IEEE, San Jose, CA (2002)
4. Amer-Yahia, S., Botev, C., Shanmugasundaram, J.: TeXQuery: A full-text search extension to XQuery. In: WWW Conference, pp. 583–594. New York (2004)
5. Babcock, B., Datar, M., Motwani, R., O'Callaghan, L.: Maintaining variance and k-medians over data stream windows. In: PODS Conference, pp. 234–243. ACM (2003)
6. Balmin, A., Hristidis, V., Papakonstantinou, Y.: Authority-based keyword queries in databases using ObjectRank. In: VLDB Conference, Toronto (2004)
7. Bar-Yossef, Z., Broder, A.Z., Kumar, R., Tomkins, A.: Sic Transit Gloria Telae: Towards an understanding of the Web's decay. In: WWW Conference, pp. 328–337 (2004)
8. Berkhin, P.: Bookmark-coloring approach to personalized pagerank computing. Internet Math. **3**(1), (2007)
9. Bhalotia, G., Hulgeri, A., Nakhe, C., Chakrabarti, S., Sudarshan, S.: Keyword searching and browsing in databases using BANKS. In: ICDE IEEE (2002)
10. Bharat, K., Bröder, A., Henzinger, M., Kumar, P., Venkatasubramanian, S.: The connectivity server: fast access to linkage information on the Web. In: WWW Conference, Brisbane, Australia (1998)
11. Borthwick, A., Sterling, J., Agichtein, E., Grishman, R.: Exploiting diverse knowledge sources via maximum entropy in named entity recognition. In: Sixth Workshop on Very Large Corpora. Association for Computational Linguistics (1998)
12. Chakrabarti, S.: Dynamic personalized PageRank in entity-relation graphs. In: WWW Conference, Banff (2007)
13. Chakrabarti, S., Agarwal, A.: Learning parameters in entity relationship graphs from ranking preferences. In: PKDD Conference, LNCS, vol. 4213, pp. 91–102. Berlin (2006)
14. Chakrabarti, S., Mirchandani, J., Nandi, A.: SPIN: Searching personal information networks. In SIGIR Conference, pp. 674–674 (2005)
15. Chakrabarti, S., Puniyani, K., Das, S.: Optimizing scoring functions and indexes for proximity search in type-annotated corpora. In: WWW Conference. Edinburgh (2006)
16. Chazelle, B.: The soft heap: an approximate priority queue with optimal error rate. JACM **47**(6), 1012–1027 (2000)
17. Cohen, E.: Estimating the size of the transitive closure in linear time. In: FOCS Conference, pp. 190–200 (1994)
18. Craswell, N., Szummer, M.: Random walks on the click graph. In: SIGIR Conference, pp. 239–246. ACM (2007)
19. Dalvi, B., Kshirsagar, M., Sudarshan, S.: Keyword search on external memory data graphs. In: VLDB Conference (2008)
20. Doyle, P., Snell, L.: Random walk and electric networks. In: Mathematical Association of America (1984)
21. Fagin, R., Lotem, A., Naor, M.: Optimal aggregation algorithms for middleware. JCSS **66**(4), 614–656 (2003)
22. Faloutsos, C., McCurley, K.S., Tomkins, A.: Connection subgraphs in social networks. In: Workshop on Link Analysis, Counterterrorism, and Privacy. SDM Conference (2004)
23. Faloutsos, M., Faloutsos, P., Faloutsos, C.: On power-law relationships of the internet topology. In: SIGCOMM, pp. 251–262 (1999)
24. Fogaras, D., Rácz, B., Csalogány, K., Sarlós, T.: Towards scaling fully personalized PageRank: algorithms, lower bounds, and experiments. Internet Math. **2**(3), 333–358 (2005)
25. Graefe, G.: Query evaluation techniques for large databases. ACM Computing Survey **25**(2), 73–170 (1993)
26. Grishman, R., Sundheim, B.: Message understanding conference-6: A brief history. In: Proceedings of the 16th conference on Computational linguistics, pp. 466–471. Association for Computational Linguistics (1996)
27. Gupta, M., Pathak, A., Chakrabarti, S.: Fast algorithms for top-k personalized PageRank queries. In: WWW Conference, pp. 1225–1226 (2008)
28. Gyöngyi, Z., Garcia-Molina, H., Pedersen, J.: Combating web spam with TrustRank. In: VLDB Conference, pp. 576–587. (2004)
29. Han, J., Pei, J., Yin, Y., Mao, R.: Mining frequent patterns without candidate generation: A frequent-pattern tree approach. Data Min Knowl Discov **8**(1), 53–87 (2004)
30. Hwang, H., Balmin, A., Reinwald, B., Nijkamp, E.: BinRank: scaling dynamic authority-based search using materialized subgraphs. In: ICDE, pp. 66–77. IEEE Computer Society (2009)
31. Järvelin, K., Kekäläinen, J.: IR evaluation methods for retrieving highly relevant documents. In: SIGIR Conference, pp. 41–48 (2000)
32. Jeh, G., Widom, J.: Scaling personalized web search. In: WWW Conference, pp. 271–279 (2003)
33. Kamvar, S.D., Haveliwala, T.H., Manning, C.D., Golub, G.H.: Extrapolation methods for accelerating PageRank computations. In: WWW Conference, pp. 261–270 (2003)
34. Kleinberg, J.M.: Authoritative sources in a hyperlinked environment. JACM **46**(5), 604–632 (1999)
35. Koren, Y., North, S.C., Volinsky, C.: Measuring and extracting proximity in networks. In: SIGKDD Conference, pp. 245–255. ACM (2006)
36. Koudas, N., Srivastava, D.: Data stream query processing. In: ICDE p. 1145 (2005)
37. Lempel, R., Moran, S.: Rank-stability and rank-similarity of link-based web ranking algorithms in authority-connected graphs. Information Retrieval **8**(2), 245–264 (2005)
38. Manning, C.D., Schütze, H.: Foundations of Statistical Natural Language Processing. MIT, Cambridge (1999)
39. McSherry, F.: A uniform approach to accelerated pagerank computation. In: WWW Conference, pp. 575–582 (2005)
40. Miller, G., Beckwith, R., FellBaum, C., Gross, D., Miller, K., Tengi, R.: Five Papers on WordNet. Princeton University (1993)
41. Minkov, E., Ng, A., Cohen, W.W.: Contextual search and name disambiguation in email using graphs. In: SIGIR Conference (2006)
42. Page, L., Brin, S., Motwani, R., Winograd, T.: The PageRank citation ranking: bringing order to the Web. Manuscript, Stanford University (1998)
43. Pan, J.-Y., Yang, H.-J., Faloutsos, C., Duygulu, P.: Automatic multimedia cross-modal correlation discovery. In: SIGKDD Conference, pp. 653–658 (2004)

44. Pandurangan, G., Raghavan, P., Upfal, E.: Using PageRank to characterize web structure. In: COCOON, LNCS 2387, pp. 330–339 (2002)
45. Pathak, A., Chakrabarti, S., Gupta, M.S.: Index design for dynamic personalized PageRank. In: ICDE, pp. 1489–1491 (2008)
46. Sarkar, P., Moore, A.W.: A tractable approach to finding closest truncated-commute-time neighbors in large graphs. In: UAI Conference (2007)
47. Sarkar, P., Moore, A.W., Prakash, A.: Fast incremental proximity search in large graphs. In: ICML, pp. 896–903 (2008)
48. Silverstein, C., Henzinger, M., Marais, H., Moricz, M.: Analysis of a very large AltaVista query log. Technical Report 1998-014, COMPAQ System Research Center (1998)
49. Sleator, D.D., Temperley, D.: Parsing English with a link grammar. In: Third International Workshop on Parsing Technologies (1993)
50. Theobald, M., Weikum, G., Schenkel, R.: Top-k query evaluation with probabilistic guarantees. In: VLDB Conference, pp. 648–659 (2004)
51. Tong, H., Faloutsos, C.: Center-piece subgraphs: problem definition and fast solutions. In: SIGKDD Conference (2006)
52. Tong, H., Faloutsos, C., Koren, Y.: Fast direction-aware proximity for graph mining. In: SIGKDD Conference, pp. 747–756. ACM (2007)
53. Tong, H., Faloutsos, C., Pan, J.-Y.: Fast random walk with restart and its applications. In: ICDM (2006)