# Dynamic Personalized Pagerank in Entity-Relation Graphs

Soumen Chakrabarti
IIT Bombay
soumen@cse.iitb.ac.in

## ABSTRACT

Extractors and taggers turn unstructured text into entity-relation (ER) graphs where nodes are entities (email, paper, person, conference, company) and edges are relations (wrote, cited, works-for). Typed proximity search of the form `type=person NEAR company~"IBM", paper~"XML"` is an increasingly useful search paradigm in ER graphs. Proximity search implementations either perform a Pagerank-like computation at query time, which is slow, or precompute, store and combine per-word Pageranks, which can be very expensive in terms of preprocessing time and space. We present HubRank, a new system for fast, dynamic, space-efficient proximity searches in ER graphs. During preprocessing, HubRank computes and indexes certain "sketchy" random walk fingerprints for a small fraction of nodes, carefully chosen using query log statistics. At query time, a small "active" subgraph is identified, bordered by nodes with indexed fingerprints. These fingerprints are adaptively loaded to various resolutions to form approximate personalized Pagerank vectors (PPVs). PPVs at remaining active nodes are now computed iteratively. We report on experiments with CiteSeer's ER graph and millions of real CiteSeer queries. Some representative numbers follow. On our testbed, HubRank preprocesses and indexes 52 times faster than whole-vocabulary PPV computation. A text index occupies 56 MB. Whole-vocabulary PPVs would consume 102 GB. If PPVs are truncated to 56 MB, precision compared to true Pagerank drops to 0.55; in contrast, HubRank has precision 0.91 at 63 MB. HubRank's average query time is 200–300 milliseconds; query-time Pagerank computation takes 11 seconds on average.

**Categories and Subject Descriptors:** H.3.1 [**Information Systems**]: Information Storage and Retrieval—*Content Analysis and Indexing*; H.3.3 [**Information Systems**]: Information Search and Retrieval—*Retrieval models*

**General Terms:** Algorithms, Experimentation, Measurement

**Keywords:** Personalized Pagerank, Graph proximity search

## 1. INTRODUCTION

Search is maturing to take advantage of taggers, annotators and extractors that associate entities and relations (ER) with text. E.g., recent personal information management systems [7] represent the extracted data explicitly as an ER graph, and enable powerful searches over the textual graph data model. A typical graph fragment is shown in Figure 1.

A very useful search paradigm that has surfaced in many forms recently [19, 20, 16, 3] is *proximity search* or *spreading activation*. E.g., a person who works in IBM on XML can be sought by issuing the "schema-light" query `type=person NEAR company~"IBM", paper~"XML"`. Note that the relation *works-for* has not been used, and we can further reduce schema information by, e.g., relaxing `paper~"XML"` to `*~"XML"`), which will also use, e.g., emails containing "XML". In general, fully offline static ranking is not feasible in this application domain, because the match predicates can be diverse, even if limited to words.


*WWW 2007,* May 8–12, 2007, Banff, Alberta, Canada.
ACM 978-1-59593-654-7/07/0005.

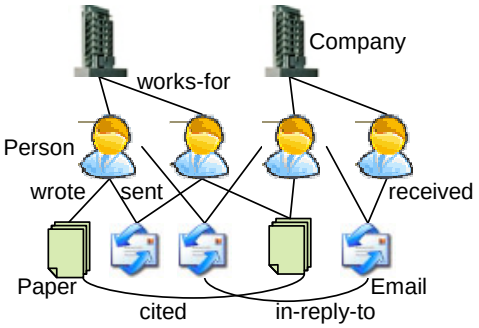


**Figure 1: A typical ER graph**

The `NEAR` operator broadly follows a personalized Pagerank-like [11, 12] semantics. A random surfer model is constructed as in Pagerank [5], with two modifications:

- The surfer does not follow edges out of a node uniformly at random. Edges have associated types; types are associated with different walk probabilities [3, 17, 6] This is critical for accuracy in ER graphs: the strengths of all relations should not be the same, and a balance must be struck between query-specific and global node prestige.
- When the surfer jumps or teleports, he jumps to a node that satisfies a match predicate, e.g., a paper containing "XML" or a company with "IBM" in the text description, and not a node uniformly at random from the whole graph.

Using standard notation, the ER graph is denoted $G = (V, E)$, the "conductance" of edge $(u, v) \in E$ is $\Pr(v|u)$, i.e., the probability that the random surfer walks from $u$ to $v$, and is written as element $(v, u)$ in a $|V| \times |V|$ conductance matrix $C$, whose columns sum to 1. $0 < \alpha < 1$ is the probability of walking (as against jumping) in each step. The teleport vector is $r$; $r(u)$ is positive only for nodes that match some query word. $r$, being a multinomial distribution, has unit $L_1$ norm. The $|V| \times 1$ personalized Pagerank vector (PPV) for teleport vector $r$ is written as $p_r$, and is the solution to

$$p_r = \alpha C p_r + (1 - \alpha) r. \quad (1)$$

We will omit $r$ when unnecessary or clear from the context.

## 1.1 The problem

Spreading activation has been proposed for searching in graphs for over a decade [19, 20, 16]. ObjectRank [3] was among the first large-scale implementations of proximity search. In ObjectRank, a PPV is precomputed for each word in the corpus vocabulary and stored in decreasing order of node scores (which means node IDs must be stored too, taking $8|V|$ bytes if `int` and `float` are used). ObjectRank supports a few monotone score-combining functions for multi-word queries. A multi-word query is executed by an efficient merge of per-word PPVs. Balmin *et al.* [3, Section 6] demonstrated, through a relevance feedback survey, that ObjectRank captured a sufficiently powerful class of scoring/ranking functions.

Vocabulary grows quickly with corpus size and can easily reach a million. Precomputing a million PPVs will take too long, even though ObjectRank uses some clever tricks to reduce PPV computation time for "almost-acyclic" graphs. The public ObjectRank demo appears to maintain a disk cache of word PPVs which are used as and when possible. If a query "misses" in cache, it is rejected and the missing PPVs are computed offline for possible later use (see Appendix A).

Cache space is the other issue. A cache of reasonable size will generally be much smaller than the vocabulary size times $|V|$. To save space, ObjectRank truncates the PPVs if a PPV element is smaller than some threshold. The effects of truncation on multi-word queries have not been thoroughly studied before, to our knowledge. In Section 2.4 we show that multi-word ranking accuracy suffers significantly if we truncate PPVs enough to match the size of a basic text index.

Personalized Pagerank (2002) was invented two years before ObjectRank (2004), but, surprisingly, there has been no open work[1] to exploit PPVs to solve the performance challenges in the ER graph search framework. Langville and Meyer write in their well-known survey [13]: *"If the holy grail of real-time personalized search is ever to be realized, then drastic speed improvements must be made, perhaps by innovative new algorithms."*

## 1.2 Our contribution

Our goal is to preprocess the ER graph must faster than computing all word PPVs, and yet answer queries much faster than a query-time ObjectRank computation, while consuming additional index space comparable to a basic text index. To this end, our key ideas, elaborated into a sketch of our system in Section 3, are as follows:

- Based on query logs, choose words *and other entity nodes* for which PPVs are pre-indexed (Section 4). These are called *hub nodes*.
- Do not compute exact PPVs, but exploit the random walk trick of Fogaras *et al.* [10] to store approximate PPVs in the form of *fingerprints* (FPs) —Section 5.
- Given a query, identify a small *active* subgraph whose PPVs must be computed, bordered by *blocker* nodes with cached fingerprints (Section 6).
- Adaptively load only a portion of the fingerprints of the blocker nodes, to save memory and computation significantly (Section 7).
- Iteratively estimate the required PPVs based on the small active subgraph, faster than running Pagerank on the whole graph, and report the top results (Section 8).

In addition, we provide many practical guidelines to exploiting partially indexed PPVs in the context of ER graph search. Both our indexing and search steps are, to some extent, "anytime algorithms" in the sense that we can abandon them at any time and get increasing quality with the effort invested.

Some indicators of HubRank performance: On our testbed, HubRank preprocesses and indexes 52 times faster than whole-vocabulary PPV computation. A text index occupies 56 MB. Whole-vocabulary PPVs would consume 102 GB. If PPVs are truncated to 56 MB, precision compared to true Pagerank drops to 0.55; in contrast, HubRank has precision 0.91 at 63 MB. HubRank's average query time is 200–300 milliseconds; query-time Pagerank computation takes 11 seconds on average.

## 1.3 Relation to earlier work

While Pagerank has been personalized in various ways since the first papers by Jeh and Widom (J&W) [12] and Haveliwala [11], hub selection is largely unexplored, especially in the context of ER graph search. Moreover, we know of no public work that uses query log statistics to pick hubs.

Fogaras *et al.* [10] not only prove that complete, full-precision personalization is doomed to use quadratic space, but they also give a simple, practical alternative: a Monte Carlo approximation of PPVs in the form of a fingerprint (FP). FPs are critical to our success, but we go farther in a few ways. First, we compute FPs only for a few, carefully-chosen nodes. Second, we adaptively control the resolution to which we compute and use various FPs. Third, because they keep FPs for each node, Fogaras *et al.* can compute a PPV estimate for node $u$ based on the FPs at only the out-neighbors of $u$. Because we may have a bigger active subgraph, we must resort to an iterative PPV computation.

Our active subgraph expansion draws from a common intuition of influence decaying with distance, because teleport deadens long-range influence [1, 9, 8]. Very recently, Berkhin [4] has independently suggested an active subgraph expansion method called the "bookmark coloring algorithm" (BCA) which is similar to our proposal, but he used PPVs, not FPs, and did not optimize the hub set using query logs. We will compare our method with BCA in Section 8.

# 2. BACKGROUND

## 2.1 Personalized Pagerank vectors (PPVs)

The basic (personalized) Pagerank recurrence is expressed in Equation (1). J&W [12] showed two far-reaching but easily-verified results that we cite below.

*Linearity.* $p_r = \alpha C p_r + (1-\alpha)r$ solves to $p_r = (1-\alpha)(\mathbb{I} - \alpha C)^{-1} r$, which is linear in $r$. Therefore, a linear combination of teleports is satisfied by the same linear combination of corresponding Pageranks:

$$p_{\gamma r} = \gamma p_r \quad \text{and} \quad p_{r_1+r_2} = p_{r_1} + p_{r_2}; \tag{2}$$

[1]Google reports four hits for the query `objectrank "personalized pagerank"` and two hits for the query `objectrank ppv`. Teoma/ExpertRank personalizes at topic level. `www.google.com/psearch` is not for ER graphs (yet).

for any scalar $\gamma$. (The above holds for *any* real $\gamma$ and any vectors $r_1$ and $r_2$, not just valid teleport vectors.)

In Section 3 we will propose a graph representation where each query word $w$ will be a node, connected to entity nodes where the word appears. Computing a PPV for the word $w$ will amount to setting a teleport vector $r = \delta_w$ in which $\delta_w(w) = 1$ and $\delta_w(u) = 0$ for all nodes $u \neq w$. Given a multi-word query, if we have available the PPV $p_{\delta_w}$ for each word $w$, we can compute the final scores of each node as a linear combination of per-word PPVs. In general for word or entity node $u$, $p_{\delta_u}$ is also called $\mathrm{PPV}_u$.

*PPV Decomposition.* With $\mathrm{PPV}_u$ defined as above,

$$\mathrm{PPV}_u = \sum_{(u,v)\in E} \alpha C(v,u)\,\mathrm{PPV}_v + (1-\alpha)\delta_u, \qquad (3)$$

or, more compactly, $Q = \alpha QC + (1-\alpha)\mathbb{I}$, where the $u$th column of $Q$ is $\mathrm{PPV}_u$ and $\mathbb{I}$ is the $|V| \times |V|$ identity matrix. The decomposition property is useful when we wish to compute an estimate of $\mathrm{PPV}_u$ from cached approximations to $\mathrm{PPV}_v$ for out-neighbors $v$ of $u$.

## 2.2 Monte Carlo fingerprints (FPs)

Fogaras *et al.* [10] proved the very important negative result that if a hub set $H \subset V$ is used, exact storage of all PPVs of $H$ will take $\Omega(|H|\,|V|)$ bits, no matter how clever a compression mechanism is devised. (If $H$ has a node for each word in the corpus vocabulary, $|H|\,|V|$ is unacceptably large.) They also showed related bounds where some error could be tolerated. Then they proposed Monte Carlo PPV estimates: instead of computing an exact $\mathrm{PPV}_u$, simulate the random surfer as follows:

1. Sample a random walk length $\lambda$ from a geometric distribution: $\Pr(\Lambda = \ell) = \alpha^\lambda(1-\alpha)$.
2. Take $\lambda$ walk steps using $C$ (see Equation (1)), ending in node $v$, say.

The walk is repeated `numWalks` times, where `numWalks` is a tuned parameter, and a frequency histogram $\mathrm{FP}_u$ over terminating nodes $v$ is created. J&W as well as Fogaras *et al.* showed that $\mathrm{PPV}_u(v) = \Pr(\text{random surfer finishes at node } v)$. For a fixed error tolerance in PPVs, a fixed `numWalks` suffices. An additional benefit is that FP computation involves largely integer arithmetic, faster than heavy floating-point computation required to solve Equation (1).

In experiments, Fogaras *et al.* computed FPs for every node, keeping `numWalks` small to maintain speed. When $\mathrm{PPV}_u$ was needed, they used the decomposition property (3) unrolled exactly once to get the benefit of FPs stored at all out-neighbors $v$ of $u$. In our case FPs are not stored at all nodes, so we must work harder to reconstruct $\mathrm{PPV}_u$ (Section 8).

## 2.3 Experimental testbed and measurements

*The ER data graph.* The CiteSeer corpus we obtained has 709173 words over 1127741 entity nodes. Our system can scale to such sizes and beyond, but query-time personalized Pagerank computation (1) is typically 30–50 times slower. For a thorough comparative evaluation of ObjectRank and HubRank, we picked papers and authors in CiteSeer prior to 1994. This subset has about 172000 words over 74000 nodes (which is more in line with the two data sets used with ObjectRank, having 13700 nodes and 12341 words and 55000 nodes and 40577 words). (CiteSeer is just one example of the text-embedded-in-graph model that is becoming ubiquitous, e.g., in personal information management [7], but obtaining real query logs would be challenging.)

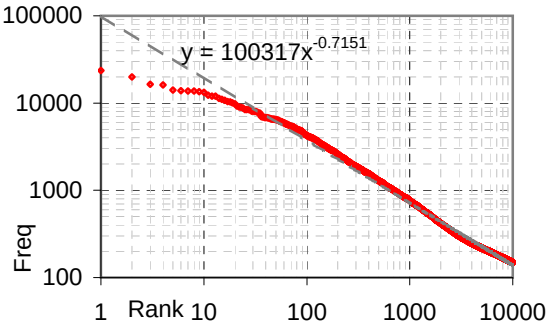


**Figure 2: Typical Zipfian distribution of word frequencies over almost two million queries.**

*The query log.* We obtained 1.9 million queries from CiteSeer, with an average of 2.68 words per query. Word frequencies are distributed in a typical Zipfian manner shown in Figure 2. We used samples of typical size 10000 from the first 100000 queries as test data for long experiments, while all but the first 100000 queries were used to train and tune our indices. This sampling procedure (unlike uniform random sampling) made sure that we are not benefiting from splitting a query session with shared words into the training and test set. To reduce longer-range dependencies our test queries are chronologically *before* training queries.

*Hardware and software.* Experiments were run on a 4-CPU 2.2 GHz Opteron server with 8 GB RAM and Ultra-320 SCSI disks. All code was written in Java (64-bit JDK1.5) and exploited the trivial parallelism across queries on all four CPUs. Unless otherwise specified, Pagerank iterations used $\alpha = 0.8$ and were stopped when $L_1$ difference between iterates dropped below $10^{-6}$.

*Comparing scores and rankings.* When evaluating accuracy, we will measure two score-related and two rank-related indicators of quality [10], comparing the test algorithm with "full-precision" ObjectRank.

**$L_1$ score difference:** If $p$ is the full-precision PPV, and we estimate $\hat{p}$, then $\|\hat{p} - p\|_1$ is a reasonable first number to check. However, it is not scale-free. I.e., for a larger graph, we must demand a smaller difference. Moreover, it is not a faithful indicator of *ranking* fidelity [14].

**Precision at $k$:** $p$ induces a "true" ranking on all nodes $v$, while $\hat{p}$ induces a distorted 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 low in the ranked list.

**Relative average goodness (RAG) at $k$:** Precision can be excessively severe. In many real-life social networks,

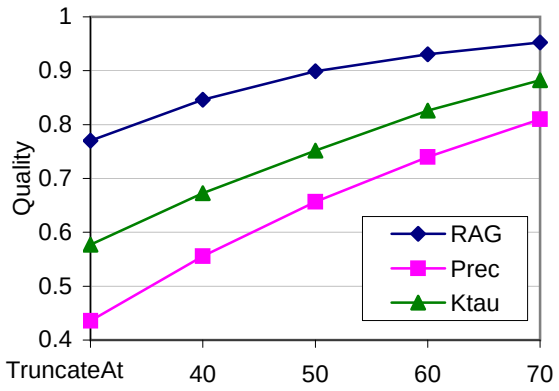


**Figure 3: Truncation reduces the ranking accuracy of ObjectRank significantly.**

near-ties in Pagerank values are common. If the *true scores* of $\hat{T}_k$ are large, our approximation is doing ok. One proposal is (note that $\hat{p}$ is not used):

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

**Kendall's $\tau$:** Node scores in a PPV are often closely tied. Let exact and approximate node scores be denoted $S_k(v)$ and $\hat{S}_k(v)$, 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 $(S_k(v) - S_k(w))(\hat{S}_k(v) - \hat{S}_k(w))$ is strictly positive, and *discordant* if it is strictly negative. 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 $\tau$ is defined as

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

(Unlike Fogaras *et al.*, we do not limit to pairs whose scores differ by at least 0.01 or 0.001, so our $\tau$ is generally smaller.)

Throughout, we use a fairly stringent $k = 100$ for evaluation.

## 2.4 ObjectRank accuracy and performance

Since ObjectRank stores PPVs sorted by decreasing Pagerank value, node IDs need to be stored explicitly. If Pagerank values are stored as `floats`, each entry requires 8 bytes, so, if all word-PPVs were stored, about 102 GB would be required. To put this in perspective, a Lucene [2] text index takes only 56 MB, which is only a 0.00056 fraction of 102 GB. For our CiteSeer subset, this means that, on average, for each word, we can store the top 41 nodes of 74000. This corresponds closely with numbers in the range of 7–84 reported in the ObjectRank paper [3].

How does truncation affect scoring and ranking accuracy, compared to the "full-precision" ObjectRank? For each query, we separately computed PPVs for each word in the query, truncated these word PPVs, then combined them (using the linearity property of PPVs, see Section 2.1).

In Figure 3 we plot RAG, precision and Kendall's $\tau$ of ObjectRank with PPVs truncated at various ranks (x-axis). 10000 queries were sampled for testing; ObjectRank took about 40 CPU-hours to complete the set, or about 14 seconds per query. The accuracy is low in this range, even for the relatively forgiving RAG measure. From Figure 3, it would appear that while truncating at 41 nodes results in poor accuracy, a few hundred nodes per word may be adequate. But this will not scale; if we tried to process the whole CiteSeer corpus with 709173 words, retaining even 100 nodes per word PPV would already consume 567 MB. More problematic is that we need to calculate all word PPVs in the first place, before we can truncate them. Otherwise, we will need to reject queries and calculate the requisite PPVs offline (Appendix A). HubRank offers a practical solution to this dilemma.

# 3. ARCHITECTURE OVERVIEW

In this section, we first describe (our adaptation of) the ObjectRank scoring model. Then we give an overview of how a query is executed; this naturally leads to hub selection and query optimization issues. These specific technical problems are solved in the rest of the paper.

## 3.1 Scoring in a TypedWordGraph

As mentioned before, the ER graph has many (node and) edge types: paper-cited-paper, author-wrote-paper, etc. Edge types are denoted $t$, taken from a small set $T$. Each edge $e = (u, v)$ has an associated type $t(e)$. Associated with each edge type $t$ is a number $\beta(t) \geq 1$. Thus the weight of edge $e$ is $\beta(t(e))$. In the preprocessing step, we index the text of every node of the ER graph. We use Lucene [2] to create an inverted index from words to entity node IDs.

A query is a set of distinct words. To process the query, the ER graph is augmented with one node for each query word, connected to entity nodes where the word appears, with special edges of type "word-to-entity". A word node appears in $W$ only if it matches at least one entity; thus, no word node is a dead end. For the moment assume that no entity node is a dead end.

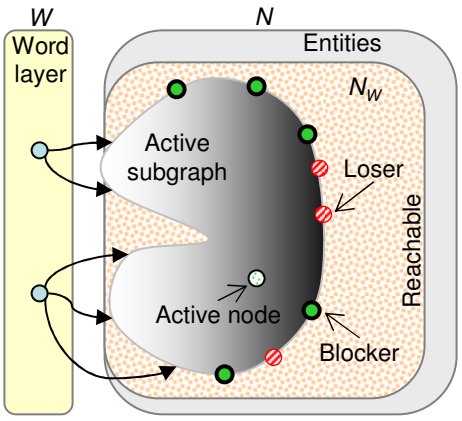


**Figure 4: TypedWordGraph with active subgraph, blockers and losers illustrated (see Section 3.2 for definition of blockers and losers).**

The *conductance* of edge $(u, v) \in E$ is now defined as

$$C(v, u) = \frac{\beta(t(u, v))}{\sum_{(u,w) \in E} \beta(t(u, w))} \quad (4)$$

Element $r(w)$ in the teleport probability vector $r$ is set to $1/|W|$ for each query word node $w \in W$. Other teleport elements are zero. One may also choose non-uniform teleport to the words, if they are not considered equally important. Equation (1) is now applied with $C$ and $r$ defined as above.

*Word rareness.* Note that an "inverse document frequency" (IDF) [18] effect is built into the design. Suppose the query has one rare and one frequent word. Each gets half the Pagerank of $d$, but the rare word passes on the Pagerank to a few entity neighbors, each getting a large share. The frequent word is connected to many entity nodes, each getting a tiny share of its Pagerank. For this reason we felt no need to experiment with different teleports to query words.

*Dead-ends and irreducibility.* A subgraph $N_W \subseteq N$ is reachable from $W$, the rest can be ignored for a specific query and our graph is effectively $W \cup N_W$. To make this irreducible and aperiodic, we add fake edges from entity nodes in $N_W$ to a sink entity node $s$. This eliminates dead ends in $N_W$. $s$ has a self-loop. The score of $s$ is ignored. Equations (1) and (4) continue to apply after these modifications, and give meaningful scores (except to $s$).

*Learning $\beta$ automatically.* Assigning weights $\beta(t)$ for each edge type $t$ might seem like an empirical art from the above discussion. Indeed, ObjectRank and related systems use manually-tuned weights. However, there has been recent progress [17, 6] in learning $\beta$ or $C$ automatically from pairwise preferences (or partial orders) between nodes. Here we will assume that $\beta$ is provided to our system by a weight learner [6] prior to indexing.

## 3.2 Query processing overview

At startup, our system preloads only the entity nodes $N$ (including the sink node $s$). This would be impractical for Web-scale data, but is reasonable for ER search applications. Only the graph skeleton is loaded, costing us only about eight bytes per node and edge. Entity graphs with hundreds of millions of nodes and edges can be loaded into regular workstations. In ongoing work, we are considering disk-based skeletons; see Section 9. Word nodes are not preloaded.

Given a keyword query to execute, we instantiate the query words as nodes in $W$ and attach each word node to the entity nodes where the word appears. This gives us a setup time not too large compared to IR engines. To answer the query, we need the PPVs of the nodes corresponding to the query words.

As shown in Figure 4, we need to work on the entity subgraph reachable from $d$ through the word nodes. A total teleport mass of 1 first reaches the word nodes, then diffuses out to $N$. Every node on the way attenuates the total outflow of Pagerank by a factor $\alpha < 1$. Therefore, we expect the effect of distant nodes on a word PPV (that we wish to compute) to be decay rapidly—indicated by the gradual shading of the active region in Figure 4.

We stop expanding the active subgraph at two kinds of boundary nodes: **blocker**[2] nodes $B \subset H$ whose PPVs have been precomputed and stored, and **loser** nodes that are too far from the word nodes to assert much influence on the word PPVs. Given we want $|B| \ll |V|$, unless we exploit loser nodes our active subgraphs will be too large.

Thus we set up some kind of a "boundary value problem": the active subgraph is bounded everywhere with blocker and loser nodes, whose PPVs remain fixed. We estimate the PPVs for the remaining active nodes, including query words nodes that are not blockers. Then we linearly combine word PPVs into the final score vector, which is then sorted to return the top answer nodes.

[2]We call hubs bordering the active subgraph "blockers" because they block the expansion.

# 4. WORKLOAD-DRIVEN HUB SELECTION

In this section we present approaches to choosing a subset of word and entity nodes that we call the *hub set*.

## 4.1 Smoothing word frequencies/probabilities

Let $W_0$ be the full corpus vocabulary and $f_w$ be the frequency of word $w \in W_0$ in the query log (after discarding words not in $W_0$). We can model the log as inducing a multinomial distribution over words and find the probabilities that maximize the likelihood of the observed log: $\Pr^{\mathrm{MLE}}(w) = f_w / \sum_{w'} f_{w'}$. (MLE is maximum likelihood estimate.) In general, even a large log will not touch all words in $W_0$, and many $w \in W_0$ will get assigned $\Pr^{\mathrm{MLE}}(w) = 0$. Given the long-tailed nature of query logs (Figure 2), this is a problem, because the above model will assign strictly zero probability of seeing a word in $W_0$ that did not occur in the log. This is a standard problem in NLP [15, Section 6.2.2], and handled by *smoothing* the word distribution. Lidstone smoothing is simple and effective: propose a smoothed probability $\widetilde{\Pr}(w) = (f_w + \ell)/\sum_{\omega}(f_\omega + \ell)$, and tune parameter $0 < \ell^* < 1$ so as to maximize the likelihood of a "held-out" portion of the query log. We omit the fairly standard details, except to summarize the benefits in Figure 17 at the end of the paper.

```
 1: initialize a score map score(u) for nodes u ∈ W0 ∪ N
 2: for each query word w ∈ W0 do
 3:    attach node w to the preloaded entity graph
 4:    let frontier = {w} and priority(w) = Pr~(w)
 5:    create an empty set of visited nodes
 6:    while frontier ≠ ∅ do
 7:       remove some u from frontier and mark visited
 8:       accumulate priority(u) to score(u)
 9:       for each visited neighbor v do
10:          accumulate priority(u) α C(v,u) to score(v)
11:       for each unvisited neighbor v do
12:          let priority(v) = priority(u) α C(v,u)
13:          add v to frontier
14: sort word and entity nodes by decreasing score(u)
```

**Figure 5: Greedy estimation of a measure of merit to including each node into the hub set.**

## 4.2 Greedy hub scoring strategy

We wish to minimize the average time taken to compute PPVs for all active nodes over a representative query mix. PPV computation time is likely to be directly related to the number of active nodes, but the connection is not mathematically predictable. Even if we were to make simplifying assumptions (such as a fixed number of iterations), the problem of picking the best subset reduces to hard problems like dominating set or vertex cover. Even quadratic- or cubic-time algorithms on CiteSeer-scale graphs may be

impractical. Therefore we turn to efficient and reasonable near-linear-time heuristics.

An indirect approach is to try to arrest active graph expansions with blocker nodes as quickly and effectively as possible. I.e., we want to pick a small number of hub nodes that will block expansions from a large number of frequent query word nodes. If a node is a good hub, it will be reachable along short paths from frequent query word nodes. This leads to the greedy hub ordering approach shown in Figure 5; it might be regarded as a fast if crude approximation to the push step in BCA [4].

## 4.3 Preliminary evaluation

Because teleport is strongest at word nodes and then diffuses out to entities with a loss incurred at every step, it may appear that the 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 Figure 6. The order returned by the algorithm in Figure 5 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.

A natural question arises here: Is the nontrivial word-entity mix essential to fast query-processing, or is it an artifact of our heuristic ordering? If latter, a suitable word PPV caching scheme associated with ObjectRank might be adequate.

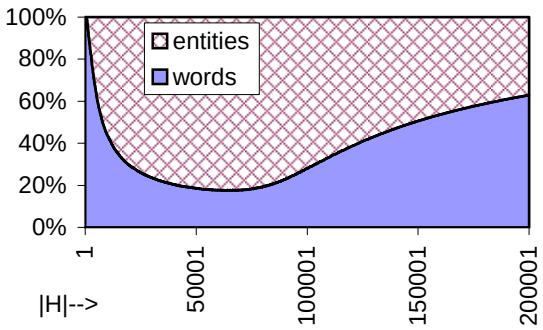


**Figure 6: For reasonable hub set sizes, entity nodes are highly desirable compared with word nodes; the best case is a nontrivial mix.**

To avoid a large number of lengthy runs with different sizes of $H$, we measure surrogates of actual running time: the number of active, blocker and loser nodes as we pick prefixes of different sizes from the ordering returned from Figure 5. (The definitions of blocker and loser are made precise in Figure 9 in Section 6.) In Section 8 we establish that these are indeed correlated to running time.

We wish to compare two orders: the mixed order returned by the code in Figure 5, and the order with all entity nodes removed. In Figure 7, we see that allowing entity nodes into the mix significantly reduces the number of active nodes and losers, and increases the number of blockers. Smaller active set is better because there are fewer PPVs to iterate over. Larger blocker set is better, because there are more

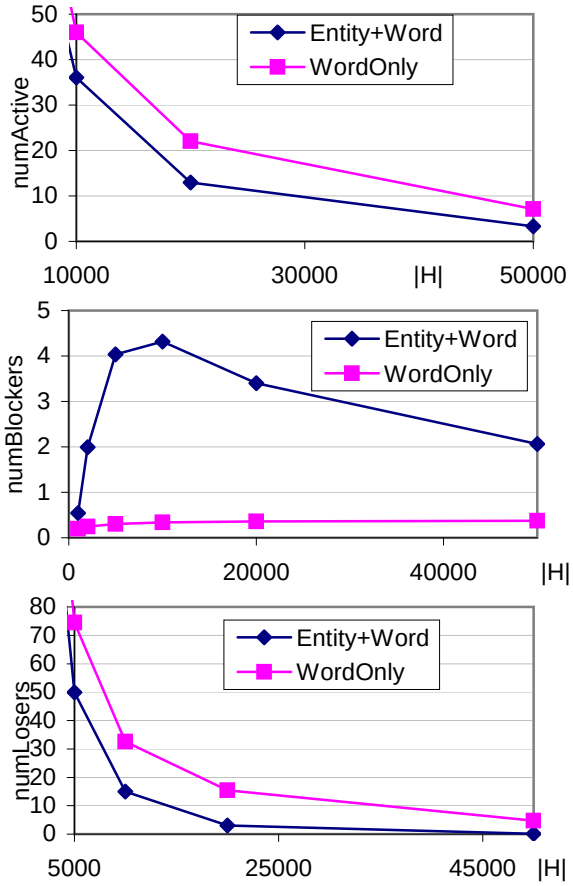


**Figure 7: Allowing entity hub nodes improves the prospects of fast, accurate PPV estimation.**

PPVs that are pinned to fixed values. Smaller loser set is better, because fewer PPVs are crudely approximated (see Section 8). Note that the number of blocker rises, then falls as the hub set size $|H|$ is increased. This is because, for large $|H|$, blockers are found *closer* to the origin nodes.

# 5. FINGERPRINT COMPUTATION

We can now greedily pick nodes with the largest merit scores, where we will compute FPs. The score associated with a node $u$ in Figure 5 reflects a combination of how often $u$ is reached from a query, and how strong the connection is. The latter factor tells us how strongly the FP of $u$ is going to affect the accuracy of answering a typical query. Intuitively, if $u$ is in the backwaters, a very low-resolution FP computed at $u$ will suffice, whereas if $u$ is a celebrity, a high-resolution FP is desirable.

In the interest of a clean, precise analysis, Fogaras *et al.* [10] used the same `numWalks` at all nodes, but, while building a system, we are at liberty to use diverse `numWalks` at different nodes. If we assume that one random walk takes roughly a fixed time, and we have a budget of a total number of walks, we should allocate more walks to celebrity nodes and fewer walks to the backwaters. A straight-forward approach is to allocate the budgeted number of walks in proportion to the score of nodes computed in Figure 5. We can allocate walks in small batches, and terminate the process whenever we run out of space, time or patience.

We tried a number of other alternatives but nothing beat this simple policy overall in terms of space, time and accuracy. The space benefit might be explained by the fact that the space required by a FP increases sublinearly with `numWalks` (Figure 8), making skewed `numWalks` more attractive than uniform `numWalks`. Most FPs are quite sparse.

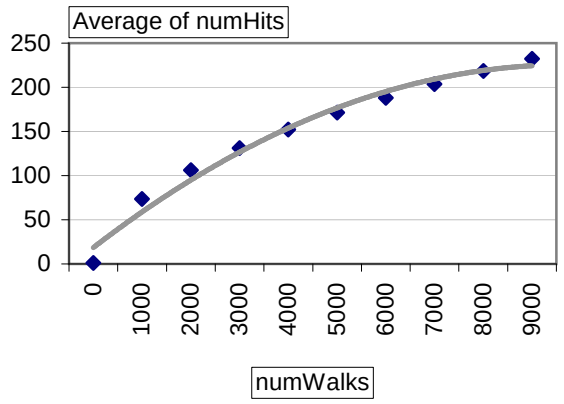


**Figure 8: The space taken by the average FP grows slightly sublinearly with increasing `numWalks`. "numHits" is the number of distinct end nodes touched by `numWalks` walks, which determines the storage required (`int+short` per-record).**

Total FP computation time was **10 CPU-hours**. Contrast this with an estimated (via word samples) **526 CPU-hours** to compute all word PPVs (even if we truncate them thereafter). Unless otherwise specified, we picked $|H| = 10000$ and average `numWalks` $= 15000$, giving a 63 MB index.

## 6. ACTIVE SUBGRAPH EXPANSION

When a keyword query is submitted, we perform an expansion process similar to that in Figure 5 to collect the active nodes $A$, except that we also identify blocker nodes $B \subset H$ whose FPs have been indexed, and loser nodes $\ell$ which are so distant from the originating node $w$ that even the largest element of PPV($\ell$) is unlikely to influence PPV($w$) much.

As in earlier work on local Pagerank computation [9, 8] we judge if PPV$_v$ is "unlikely to influence" PPV$_u$ much via a heuristic. Ideally, we should check if the conductance from $u$ to $v$ is small, but that amounts to solving part of the PPV estimation problem (which is what BCA [4] does). Chien *et al.* [9, Section 3.1] propose a one-pass weight-dissipation type algorithm similar to ours, except that, in the interest of speed, we further omit conductance via multiple paths, noting only the *largest conductance* path from $u$ to $v$. (We use all edges while iteratively computing active PPVs.) Figure 9 shows the first stage of query processing: collecting the active subgraph.

## 7. DYNAMIC RESOLUTION FP-LOADING

There is one more critical hurdle to negotiate before our basic idea works out. The big advantage of regular Pagerank/ObjectRank is that only one Pagerank vector needs to be computed, whereas we must iteratively estimate PPVs

```
 1: Input: word nodes W, abandon threshold δ
 2: Outputs: active nodes A ⊂ V', blockers B, losers L
 3: let frontier be a max priority queue
 4: insert ⟨w, 1⟩ into frontier for each w ∈ W
 5: while frontier is not empty do
 6:   remove ⟨u, s⟩ from frontier
 7:   if u ∉ A then
 8:     add u to A
 9:     if fingerprint FP_u is found in index then
10:       add u to B
11:       load(FP_u, s, δ) for blocker u
12:     else if s < δ (abandon threshold) then
13:       add u to L
14:       load a trivial FP for loser u
15:     else
16:       for each child v of u do
17:         add ⟨v, s Ĉ(v, u)⟩ to frontier
```

**Figure 9: Query-time active subgraph expansion. For load routines see Section 7.**

at all active nodes. If we convert the cached FPs into full-length PPVs and compute full-length PPVs all over the active subgraph, the sheer handicap we will face by way of floating point operations will forestall any substantial gains from HubRank.

The key trick is to extend the pruning action of step 12 in Figure 9, from discarding whole FPs to discarding parts of FPs as well. FPs are stored using Berkeley DB; the key is a node ID and the data is a sequence of (*numHits*, *nodeID*) records, sorted in decreasing order of *numHits*. As we scan down the list, we keep a cumulative hit count *sumHits*. At any point during the loading scan load(FP$_u$, $s$, $\delta$), if we find a node $v$ for which

$$s \frac{numHits(v)}{sumHits} < \delta,$$

we abandon the rest of FP$_u$, and rescale the loaded prefix to be a PPV estimate with unit $L_1$ norm.

Note that FPs are stored to diverse resolutions in the first place, but that is wrt an aggregated query mix; for a specific new query, we may need to load them to a very different set of resolutions.

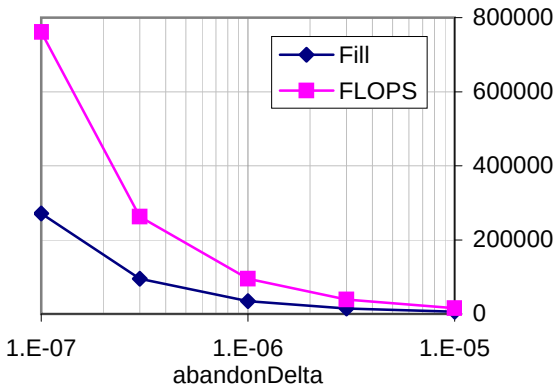


**Figure 10: Modest values of $\delta$ suffice to dramatically reduce the average "fill" (nonzero element count) of FPs loaded over all active nodes, and the number of floating-point operations per iteration.**

This dynamic pruning has dramatic effect on keeping the loaded PPVs sparse, as can be seen from Figure 10. With-

out this trick, we would not be able to beat ObjectRank by a large margin consistently. Luckily, as we shall see in Figure 13, $\delta$ has minimal effects on accuracy over a broad range. Loading FPs for a loser node $v$ is easy: we just initialize the PPV to $x_v$. We do the same for active nodes, but they are then free to float, while loser PPVs are pinned.

## 8. ITERATIVE PPV APPROXIMATION

Once the active subgraph is set up, we run the "dynamic programming" version of J&W's PPV estimation algorithm, keeping blocker and loser PPVs fixed. This just boils down to iteratively invoking Equation (3) as an iterative assignment:

$$\widehat{\mathrm{PPV}}_u^{(i)} \leftarrow \sum_{(u,v)\in E} \alpha C(v,u)\, \widehat{\mathrm{PPV}}_v^{(i-1)} + (1-\alpha)\delta_u. \qquad (5)$$

A randomized ordering of $u$s worked best.

*Convergence.* J&W prove that iterative PPV updates will converge. It follows that if we pin blockers to *true* PPVs, active PPVs will converge to true values. However, in our case, we fetch from disk $\mathrm{FP}_u$, which is a fairly crude estimate of $\mathrm{PPV}_u$, which we further truncate. Because FPs at different blockers were obtained via different random walks, they may be potentially inconsistent. We argue in Appendix B that given a set of blocker FPs, there exists a PPV assignment to active nodes consistent with the FPs, and iterative PPV updates will take us there. Unfortunately, unlike the elegant single-FP case analysis of Fogaras *et al.*, we cannot prove that at convergence we get unbiased or otherwise mathematically meaningful PPV estimates; this is left to experimental evaluation. Figure 11 validates over four (of millions of) queries that convergence is never a problem in practice.

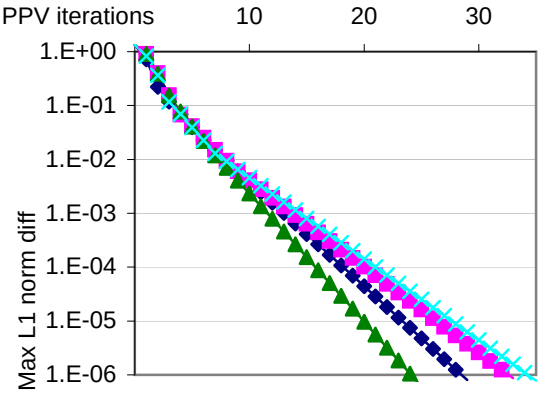


**Figure 11: Fast convergence of active PPVs.**

*Running time.* Figure 12 plots a scatter of time-to-converge against the number of active nodes. Over a wide range, running time is strongly related to the number of active nodes. This validates our hub selection approach in Section 4.3 and corroborates Figure 7.

*Effect of $\delta$ on overall speed and accuracy.* Clearly $\delta$ reduces the memory footprint and computational load of HubRank, but the critical question is, how accurate is the resulting ranking? And can that accuracy be obtained while

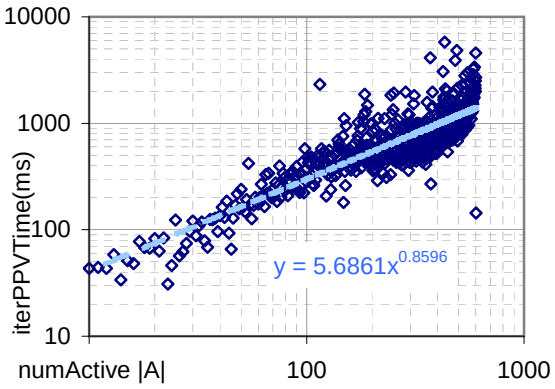


**Figure 12: Over 2–3 orders of magnitude, the time for iterative PPV estimation is almost linear in the number of active nodes.**

beating ObjectRank time by a substantial margin? Figure 13 shows that, as $\delta$ is increased, the overall time taken by HubRank drops dramatically, basically tracking Figure 10. In contrast, all accuracy indicators remain essentially flat. The ranking stability persists across 100× increase in $\delta$ and a 29-fold decrease in FP footprint (note the x-axis is a log-scale). (We found $\delta = 3 \times 10^{-6}$ the best value for our testbed.) In contrast, observe in Figure 3 how ranking quality drops sharply on a *linear* x-axis representing index size.

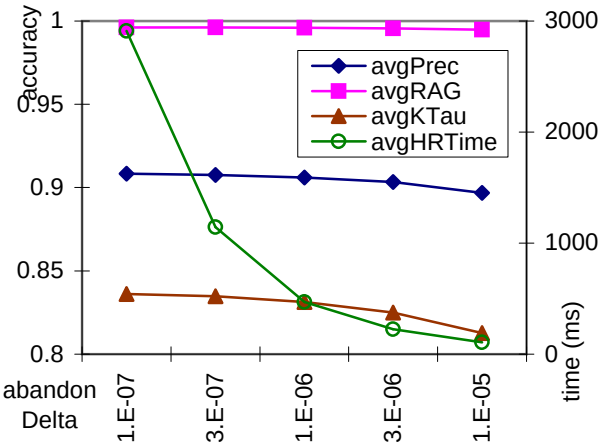


**Figure 13: HubRank accuracy and running time as a function of $\delta$, the threshold for abandoning FPs.**

Figure 14 presents average HubRank and ObjectRank query processing times in one chart, against the number of words in a query. HubRank time is more jittery, but, for short queries, some **20–40 times** faster than ObjectRank computed at query time. The gap is most pronounced at the very frequent short (1–3 word) queries.

*Effect of increasing cache size.* Average query time for HubRank drops steeply as the FP cache size increases. For our testbed, the steep decrease happens right around the same size as the Lucene text index (Figure 15). But long before the "knee" is reached, HubRank times are less than $\frac{1}{12}$th that of ObjectRank.

As the FP cache is enlarged, it accommodates more FPs with smaller `numWalks`, as per our fingerprint computation

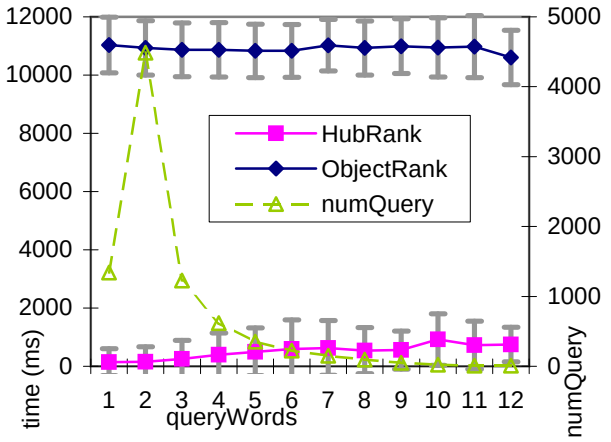


**Figure 14: HubRank and ObjectRank query times (average and standard deviation) and relative query frequency against the number of words in a query.**

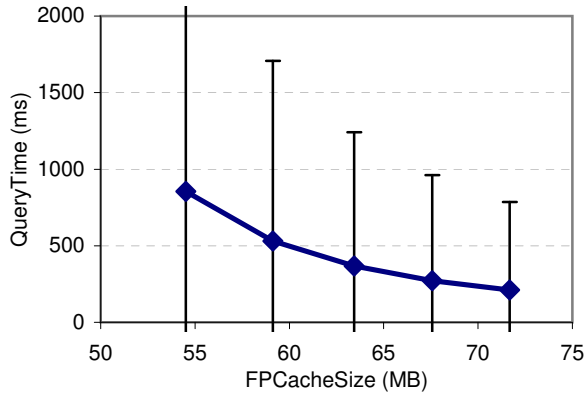


**Figure 15: Effect of cache size on HubRank query execution time (average and standard deviation).**

policy in Section 5. As shown in Figure 16, this does very modest damage (less than 0.5%) to precision and $\tau$, even as query processing time drops by over a factor of 4.

*Smoothing.* Figure 17 shows the beneficial effects of workload-smoothing on accuracy and speed. Smoothing ensures that hubs are picked reasonably close to word nodes that do not even appear in the training query log. This improves both speed and accuracy.

*Comparison with BCA.* BCA [4] can be regarded as a refined version of Figures 5 and 9. To maintain precise guarantees, BCA starts with a *residual* of $r(w)$ at word node $w$, and progresses using *push* steps. A push from node $u$ transfers $1-\alpha$ times its current residual to its score, and the remaining $\alpha$ fraction of its current residual to its out-neighbors' residuals. BCA has to maintain a heap for the node residuals. Using a Fibonacci heap, BCA is somewhat slower than HubRank in our preliminary experiments (Figure 18), but both are substantially faster than ObjectRank.

## 9. CONCLUSION

*Summary.* We presented HubRank, a workload-driven indexing and dynamic personalized search system for ER graphs.

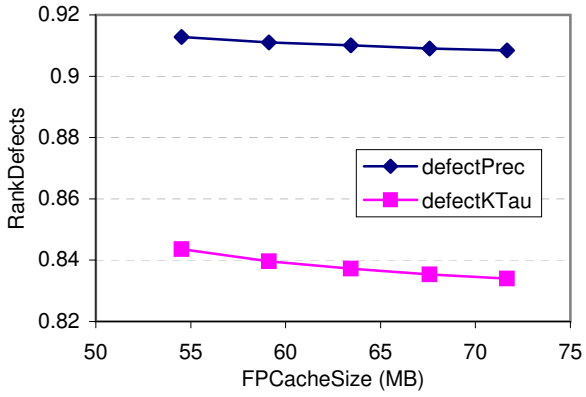


**Figure 16: As cache size increases, we include lower quality FPs, but the drop in ranking accuracy is very modest.**

| #words→ | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| NoSmoothTime | 622 | 566 | 756 | 994 |
| SmoothTime | 280 | 310 | 549 | 836 |
| NoSmoothPrec | .81 | .85 | .85 | .85 |
| SmoothPrec | .85 | .91 | .92 | .91 |

**Figure 17: HubRank time and precision with and without smoothing, at $\delta = 10^{-6}$.**

Our index preprocessing is 52 times faster than ObjectRank; our index is comparable to a text index and 0.056% the size of a full ObjectRank index. Our query time is 20–40 times smaller than query-time ObjectRank. Our indexing and search are "anytime algorithms" in the sense that we can abandon them at any time but get increasing quality with the effort invested. At present HubRank scales to the size of DBLP and CiteSeer, with several millions nodes and almost a million words.

*Failure analysis.* We separated sample queries where all words were blocked (empty active set, complete word FPs loaded) vs. queries with nontrivial active sets. Ranking quality in these two query categories were essentially identical wrt precision and RAG, but surprisingly, wrt $\tau$, empty active sets are worse (Figure 19)! This hints that limited precision of word FPs may be a significant impediment to higher accuracy; iterative PPV calculation is not too crude

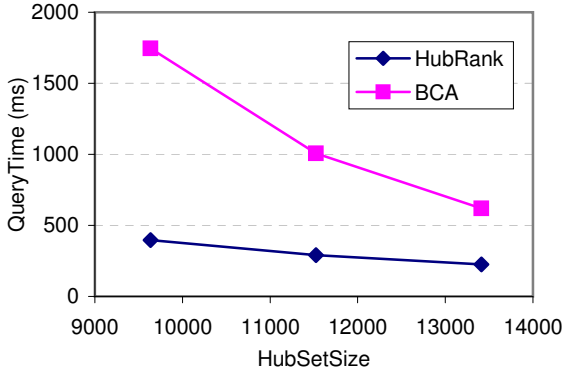


**Figure 18: BCA has somewhat higher overhead than HubRank.**

| Active subgraph | #queries | Avg RAG | Prec Prec | Avg $\tau$ |
|---|---|---|---|---|
| Non-empty | 3413 | .996 | .864 | .801 |
| Empty | 5079 | .986 | .878 | **.742** |

**Figure 19: Ranking quality in empty and non-empty active subgraphs.**

in comparison. We should revisit the proportional budget allocation policy of Section 5. The active set overflows RAM once in $\sim$ 2000 queries owing to a dearth of blockers; we need sentinel blockers or a reliable ObjectRank fall-back.

*Unresolved issues and ongoing work.* We would like to give a theoretical guarantee of the score quality similar to Fogaras *et al.*. In view of Lempel and Moran's results [14], it is unclear if we can give any guarantee of *ranking* quality in the Pagerank framework. We are working on graph clustering and indexing techniques to reduce disk seeks while expanding the active subgraph and loading blocker PPVs, while the graph is largely on disk. We would also like to support broader classes of predicates on nodes, perhaps involving structured attributes and cached views over and above word matches. We would like to report top-$k$ without evaluating the whole active set to convergence.

*Acknowledgments.* Thanks to C. Lee Giles for CiteSeer data, to Manish Gupta and Amit Phatak for cleaning the data, to Yannis Papakonstantinou for access to ObjectRank source code, and Pavel Berkhin for discussions.

## 10. REFERENCES

[1] S. Abiteboul, M. Preda, and G. Cobena. Adaptive on-line page importance computation. In *WWW Conference*, pages 280–290, 2003.

[2] Apache Software Group. Jakarta Lucene text search engine. GPL Library, 2002.

[3] A. Balmin, V. Hristidis, and Y. Papakonstantinou. Authority-based keyword queries in databases using ObjectRank. In *VLDB*, Toronto, 2004.

[4] P. Berkhin. Bookmark-coloring approach to personalized pagerank computing. *Internet Mathematics*, 3(1), Jan. 2007. Preprint.

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *WWW Conference*, 1998.

[6] S. Chakrabarti and A. Agarwal. Learning parameters in entity relationship graphs from ranking preferences. In *ECML/PKDD*, volume 4213 of *LNCS*, pages 91–102, Berlin, 2006.

[7] S. Chakrabarti, J. Mirchandani, and A. Nandi. SPIN: Searching personal information networks. In *SIGIR Conference*, pages 674–674, 2005.

[8] Y.-Y. Chen, Q. Gan, and T. Suel. Local methods for estimating pagerank values. In *CIKM*, Washington, DC, Nov. 2004.

[9] S. Chien, C. Dwork, R. Kumar, D. R. Simon, and D. Sivakumar. Link evolution: Analysis and algorithms. *Internet Mathematics*, 1(3):277–304, 2003.

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

[11] T. H. Haveliwala. Topic-sensitive PageRank. In *WWW Conference*, pages 517–526, 2002.

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

[13] A. N. Langville and C. D. Meyer. Deeper inside PageRank. *Internet Mathematics*, 1(3):335–380, 2004.

[14] R. Lempel and S. Moran. Rank-stability and rank-similarity of link-based web ranking algorithms in authority-connected graphs. *Information Retrieval*, 8(2):245–264, 2005.

[15] C. D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, 1999.

[16] M. Marchiori. The quest for correct information on the Web: Hyper search engines. In *WWW Conference*, Santa Clara, CA, Apr. 1997.

[17] Z. Nie, Y. Zhang, J.-R. Wen, and W.-Y. Ma. Object-level ranking: Bringing order to Web objects. In *WWW Conference*, pages 567–574, 2005.

[18] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.

[19] J. Savoy. Bayesian inference networks and spreading activation in hypertext systems. *Information Processing and Management*, 28(3):389–406, 1992.

[20] J. Savoy. An extended vector processing scheme for searching information in hypertext systems. *Information Processing and Management*, 32(2):155–170, Mar. 1996.

## APPENDIX

## A. ObjectRank CACHE MISS EXAMPLE

Output captured from `http://teriyaki.ucsd.edu:9099/examples/jsp/objrank/objectRank05.jsp` on 2006/11/12:

```
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.
```

## B. PPVS IN TypedWordGraph

Let $\mathrm{PPV}_u$ be the $u$th 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 $q_U$, the unknowns and $q_K$, the known PPVs (from blocker and loser FPs). Let $C$ be correspondingly partitioned into $\begin{bmatrix} C_{UU} & C_{UK} \\ C_{KU} & C_{KK} \end{bmatrix}$. As far as our PPV iterations go, because we never look beyond blockers and losers, only $U \to K$ and $U \to U$ edges matter; thus, we are looking for a solution to

$$q_U = \alpha\, q_U\, C_{UU} + \alpha\, q_K\, C_{KU} + \mathrm{const}_{1\times|U|}$$

but $\alpha\, q_K\, C_{KU}$ is a fixed row vector as well, so the recurrence simplifies into $q_U = \alpha\, q_U\, C_{UU} + c$, where $c$ is some fixed $1 \times |U|$ row vector. Because $\alpha\, C_{UU}$ is strictly substochastic, $(\mathbb{I} - \alpha\, C_{UU})^{-1}$ exists and so there is a unique solution for $q_U$. J&W's proof of convergence of power iterations at a rate of $\alpha$ or better can be extended; we omit the details. Unfortunately, $q_U$ is not guaranteed to be statistically meaningful (e.g., unbiased).