Towards Distributed Social Search Engines

Josep M. Pujol  
Telefonica Research, Barcelona, Spain  
jmps@tid.es

Pablo Rodriguez  
Telefonica Research, Barcelona, Spain  
pablorr@tid.es

ABSTRACT
We describe a distributed social search engine build upon open-source tools aiming to help the community to take back the Search. Access to the Web is universal and open, and so the mechanisms to search should be. We envision search as a basic service whose operation is controlled and maintained by the community itself. To that end we present an alpha version of what could become the platform of a distributed search engine fueled by the shared resources and collaboration of the community.

1. INTRODUCTION
Poropine (PQ) intends to serve as the cornerstone of an alternative approach to search that puts the emphasis on distribution and on the social dimension of their users. PQ is inherently social. Crawling, indexing, and ranking of the Web is carried out collaboratively as users browse and search without their explicit intervention. Not requesting feedback from the users frees us of the main problem of collaborative systems: lack of collaboration fades out since collaboration have zero cost. PQ does not rely on explicit feedback to operate, therefore, it remains totally transparent to the user. We believe transparency is a major feature. First because it lowers the entry barrier for those people who do actively engage in collaborative systems. Second, because it does not interfere with those users who already have habits that do not wish to change. A user should not learn PQ, but PQ should learn from her and share her snippets of knowledge and experience among her social circle while providing privacy and anonymity. PQ only considers collaboration among friends or acquaintances, which are defined by the user’s social network. This limits the accessible Web by means of searching, but the loss of recall comes with an increase of personalization, context awareness and byproducts such as serendipity and social glue. Another reason to limit the search to your social network is for efficiency and scalability as it limits propagation. PQ is distributed. The infrastructure required for a search engine already exists in the form of thousands of idle desktops and extensive residential broadband access. In PQ everyone runs their own local search engine. This distributed setting is not only more environmentally and economically sustainable. But it also leverages the end-user computing power to carry out functionalities such as contextual filtering or online recommendation, which would be expensive to scale in a centralized architecture.

2. PQ’S ARCHITECTURE
Figure 1 depicts the architecture of PQ. There are 3 independent components that can run either in the user’s desktop or distributed across several machines. The pqbar is a Firefox addon, which monitors browser events to detect browsing, searching as well as user actions while browsing (used to infer the page’s relevance). The relation between pqbar and pqnode is N:1 as the user might access the Web from different machines. The pqnode is the personal search engine of the user. It is a Ruby process composed of a mongrel servlet for communication, the HProcot based parser, the Ruby Ferret indexes for pages and search sessions and the social network module. This module obtains the social network from different sources such as Facebook, Twitter and email accounts. It aggregates all sources into a single ego-network and calculates the community structure to find clusters. The trusted proxy component, also implemented in Ruby, consists of a mongrel servlet and the modules responsible of forwarding search requests and aggregating results. All communication between components is done via WS API with parameters encapsulated in JSON. The relationships between components is better described with the following two use cases.

2.1 Feeding your PQNode
Every time a user browses a page, the pqbar will send the url to the pqnode which will wget the url, parse the content and index it on the Ferret page index. When the url is not a standard web page but it is the result of a query (detected by the pqbar via regular expressions) the pqbar will start a search session. Once completed, the pqbar sends to the pqnode the query and the set of results browsed by the user. This set contains the order of visits, time, depth, navigation graph and the actions performed on the web page. Once the search session is sent to the pqnode it is stored in the search index. We mentioned that pqbar also reports the actions performed by the user on a web page. These actions range from trivial commands such as print, save, control+c to more sophisticated and less noisy estimators of the user’s interest on the page such as adding it to Firefox’s bookmarks.
or sending it to sites such as Delicious or Reddit. *PQbar* can detect by checking Firefox’s load event and analyzing the GET request parameters. Whenever the user sends a link to a service such as social bookmarking, online social networks, news aggregator, etc., the *pqbar* takes notice. Note that the scraping of the page content is performed by *pqnode* instead of the *pqbar*. This introduces some network overhead but it guarantees that no private content rendered in the browser will ever be indexed.

PQ users contribute to the system without work overhead and in a total transparent way. Content in PQ is indexed in real time. As soon as one user in your social circle sees a page, the page will be immediately available to you. How much information can be collected by users? As moderate PQ user collected 61Mb worth of data, 4863 unique web pages and 1640 queries, after 3 months of use. A crawling robot is able to index that much in a matter of hours. However, pages indexed by the user’s *pqnode* are actual web pages that were interesting to her. Additionally, the pages indexed by the user might not be visible for the robot for a while. Transmission of links take place in many channels that robots do not have access to, e.g. emails, instant messages, facebook post and tweets to mention just a few.

### 2.2 Searching the Web through PQ

Search in PQ can be triggered either by querying directly your *pqnode* (e.g. via PQ web page) or — if the embedded search is enabled — by submitting a query to a search engine (e.g. Google, Yahoo, isoHunt, etc.). In the embedded case, results will be displayed side by side to the original search engine as depicted in Fig. 2. Search in PQ consists in:

**Local search** Each *pqnode*, upon receiving a search request, queries the Ferret page index. The TDIDF scores returned by Ferret are factored out as they are noisy due to our basic scrapping and the to the limited user’s corpus that affects the IDF calculation. The basic relevance score of the page is a function of the hits on the snippets, the number of visits and its freshness. Additionally, the query is also submitted to the query index to retrieve the top-20 similar queries. We check if an url from previous search results is also part of the current result set. If the url is also the quality result for a past query then the page’s score is increased according to the distance between queries. We can estimate the quality of a search result because the search index records the browsing pattern, time and the actions of the user during that search session. The distance between the current query and the top-5 similar queries is used to assess the user expertise on the query.

**Search propagation** The search propagation is managed by the trusted proxy that receives the query, the social network and the community partition from the *pqnode*. The proxy uses the community partition to differentiate between the clusters within the *pqnode*’s social network. The proxy queries a random sample of the neighboring nodes (up to 20 request) and then targets the cluster that returned more hits with the remaining 20 requests. The results are finally aggregated and sent back to the *pqnode* that cannot identify which *pqnode* provided a particular result.

**Ranking** Upon receiving the results from the proxy, the *pqnode* ranks them using a basic collaborative filtering (using the expertise and the relevance score) weighted by the popularity and freshness of the page in the set of results from your social network. Finally, the results are sent back to the *pqbar* for display. At this stage, computationally intensive processes could be performed without an impact on the performance of the system. For instance, results could be clustered or rerank according to the content of other open web pages to introduce some level of context.

### 3. PRIVACY AND ANONYMITY

Unlike the draconian ToS of many systems that handle user’s data, PQ operates in two simple premises. 1) The data belongs only to the user and she has absolute control over it. Therefore, we do not use aggregated data. The *pqnode*’s role is to encapsulate and to isolate the data to guarantee privacy. And, 2) all contributions to the community are anonymous unless stated otherwise by the user. The search request operation could be used be a malicious agent to probe the content of other *pqnodes*. To avoid this misbehavior that would jeopardize privacy we enforce communication by proxy, although there are other alternatives.

### 4. DEPLOYMENT AND SCALABILITY

PQ is designed to require minimal infrastructure, the *pqnode* computation, storage and bandwidth is carried out in the user’s computer. Since search only propagates one step further thanks to the limits introduced by the social network, latency is kept on check. The *pqnode*’s resources footprint is small (given that is a Ruby process). For the user mentioned before the local search takes between 0.05 to 0.2 seconds depending on cache misses. The network traffic overhead is small, the worse case scenario with 100 queries per day with a full neighborhood (40) would have an impact of 66MB/day or an average bandwidth reduction of 0.4KB/s. The only limiting factor of the *pqnode* would be its memory footprint of 60 to 100MB. Ruby is quite a memory hog. After starting *pqnode* the process already allocates 48MB, 90% of them due to the mongrel servlet. Using ruby for the *pqnode* reference implementation in a stable release should be seriously reconsidered. The trusted proxies only operate on the control plane so few servers can host enough proxies for tens of thousands users. The cost of the proxy solution represents a marginal fraction of the cost of a hosting the data plane. Furthermore, proxies could be hosted by normal users provided they were trusted by the rest of the community.

**Current deployment.** Since the auto-update feature of the *pqnode* is not yet available we are temporarily hosting the user’s *pqnodes* processes on the PQ cloud at Amazon EC2. However, this is only a temporal solution until the auto-update feature of *pqnode* is released. The temporal setting, however, poses a scalability problem due to the large memory footprint of the *pqnodes*. Although this problem will be reduced once the users’ *pqnode* processes are back to the users’ computers, it will not completely disappear. A non trivial number of users will suffer from low availability and suboptimal network connections. Offering a cost-wise solution to these users is perhaps our biggest challenge.

### 5. CONCLUSIONS

We presented that anatomy of Poropine, a working prototype of a distributed social search engine targeting to take the search back to the people. This search engine is far from finished, and hopefully it will never will as the community engages on extending it.
Figure 1: Architecture of PQ

Figure 2: Results of the search "www developer" to Google and PQ embedded search. The search is replicated automatically by the pqbar whenever the user searches in a search engine (not only Google). Results from PQ come from past searches and browsing of member of her community and herself. If the PQ had no results, the system would learn from the current search session, and consequently, another user of the same community would find relevant results on the next query.