The Case for a New Web Search Architecture

Paul Gauthier, Steven D. Gribble, Brian Bershad
Department of Computer Science & Engineering
University Of Washington
{gauthier,gribble,bershad}@cs.washington.edu

Abstract

In this paper we make the case for microsearch, an architecture in which Web search capabilities are added to a user’s local PC. We demonstrate that the substantial storage and computing capacity of a user’s local computer can be used to index the World Wide Web directly on the PC. Compared to traditional purely centralized search, microsearch enables a new class of search-based applications, improves the relevancy of results through personalization, enhances privacy, permits disconnected operation, and allows for customized search strategies. We describe an implementation of a microsearch system, several new applications, and the future implications of tomorrow’s larger storage capacities on microsearch.

1 Introduction

The architecture of Web search has remained mostly unchanged since the first cluster-based search engines of the mid-1990s. These systems resolve queries for millions of users by means of a massive cluster that caches and indexes the entire Web. Recently, technology trends have brought substantial storage capacity, on the order of hundreds of gigabytes, to an individual PC. These trends suggest that it should be possible to implement Web search in a localized fashion, where a single PC serves the queries of its owner by means of a private, locally maintained, Web index. We call such a search engine a microsearch engine.

This paper demonstrates that a modern computer is already capable of hosting a microsearch engine, and as a result, we can augment operating systems to provide applications with a locally serviced Web search primitive. Since a microsearch engine serves a single user and has the full resources of a PC at its disposal, it offers a number of advantages relative to its centralized counterparts, including tailoring search results to the interests of its user, and having the ability to handle a substantially higher per-user query load than centralized search engines can afford. This, in turn, enables a new class of applications that were previously infeasible.

We have constructed a prototype implementation of a microsearch engine and a set of applications that demonstrates its power. After describing and evaluating these, we quantify the expected impact of technological trends on microsearch, arguing that in the near future, a microsearch engine might even be able to replace its centralized predecessors.

1.1 Evolving Search

It is not uncommon for a system to evolve from a large, shared, centralized installation to a small, private, personal one. The replacement of time-shared computers with personal ones is one significant example. Personal computers offered users more autonomy and control, and enabled novel capabilities that ultimately eclipsed their centralized ancestors. Like the shift away from time-sharing, the shift from centralized search to microsearch creates new opportunities for users and applications.

Search can be enabling. By giving each user their own private search engine, new applications become feasible. For example, we have built a service which uses context from the user’s recent Web surfing history to suggest in real-time Web content that is relevant to their current desktop activity. By analogy, the shift away from time-sharing systems to PCs enabled the development of rich user interfaces that would have been impractical in a time-shared environment.

Search can be more relevant. By allowing search to evolve from a global, centralized service, to a personal, local one, it may become possible to give the user a better searching experience. In this paper, we use a personal PageRank algorithm [29], which ranks pages based on the user’s interests, to improve the relevance of search results.

Search can be private. A user’s microsearch query remains private, since it is serviced on the user’s PC and
need not be reflected to a central server, log, or database. In the move away from the time-sharing model, the ensuing privacy resulted in an abandonment of usage policies, centralized accounting and charge-back schemes, and hastened the integration of computing into offices and households.

**Search can be disconnected.** By eliminating their dependency on centralized search, users become free to submit queries from partially or completely disconnected environments. For anyone who has ever used their laptop on an airplane, the time-sharing analogy is obvious.

**Search can be customized.** A microsearch engine can be specialized to a given user, application, or search algorithm. For example, each deployed microsearch engine could independently choose which ranking algorithm to use. As well, because a microsearch runs on behalf of a single user, it can use algorithms such as HITS [25] or latent semantic indexing [16] that do not scale well to millions of users. In terms of our time-sharing analogy, personal computers allowed users to customize OS parameters like scheduling policy and cache size to the needs of their specific applications—something that was difficult in a time-sharing environment.

These potential benefits all derive from the fact that a microsearch engine sits in close proximity to the user and can devote all of its computational and storage resources to that user. The close proximity gives the microsearch engine the capability to observe the user’s activities and discover their interests so as to better resolve queries.

### 1.2 Challenges

A microsearch engine faces a number of challenges, many of which are related to its reduced storage capacity relative to a large-scale cluster. A centralized search engine has the storage to cache and index the entire Web—literally billions of pages. Although a modern PC has large amounts of disk storage, it can only afford to index just a few tens of millions of pages. Clearly, the implication of this is that a microsearch engine index can not cover the entire Web.

In spite of the fact that a PC has orders of magnitude less storage than centralized search clusters, we show that a microsearch engine running on a PC can provide high-enough quality search results to enable a class of interesting applications. Furthermore, as storage capacity trends continue to improve, the quality of microsearch engine results will improve along with it, enabling even more aggressive search applications.

Microsearch also invokes the historical transition from time-shared to personal computing systems. Although a time-shared system had substantially more computing and storage capacity than a PC, the potential advantages of the PC compensated for the commensurate degradation in raw computing power. Over time, the advantages of the PC led to its acceptance *despite* its early disadvantages in terms of mips and megabytes, as it enabled a new style of computing that was previously infeasible. Eventually, though, the core microprocessor in the personal computer became powerful enough to handle even demanding applications, eliminating that final barrier to acceptance.

We expect the adoption of microsearch to follow a similar path. Comparatively, as we show, its performance relative to a centralized search engine is acceptable, although not yet as good for reasons that are easy to understand and easy to fix. Given the anticipated improvements in storage technology, we believe that microsearch will one day be able to replace centralized search engines as the dominant search architecture.

### The rest of this paper

In the next section we discuss related work. In Section 3, we describe a prototype microsearch engine implementation. Using this prototype as a platform, we have implemented several novel applications, which we discuss in Section 4. We examine technology trends and how they will impact the design and capabilities of future microsearch engines in Section 5. Finally, in Section 6 we conclude.

### 2 Related Work

Today, centralized Web search engines consist of tens of thousands of machines, indexing billions of pages and answering thousands of queries per second. The origins of Internet-scale search, though, can be traced back to Archie [17] and WAIS [23], both of which provided textual search over documents made accessible through FTP or Gopher [10]. The accessibility and rich interface of the Web caused it to quickly eclipse these systems, leading to the emergence of large-scale Web spidering and search.

The first widely known, crawler-based, full-text indexing Web search engine was WebCrawler [30]. At the time of publication, WebCrawler indexed nearly 50,000 documents from over 9,000 servers, and handled 6,000 queries per day. Following WebCrawler, a number of increasingly large-scale search engines were commercialized and popularized, beginning with Inktomi [11], one of the first to achieve scalability using clustered computing, and including Lycos [5], AltaVista [28], and Google [12]. There are several pieces of non-commercial Web search software, including Apache Lucene [19] and Nutch [13], on which our prototype is built.

In terms of privacy, a microsearch engine can resolve search queries without releasing information to a centralized service, where it can be stored, processed, or even
released to third parties, such as advertisers, competitors, or the government. Previous work has demonstrated that queries to a centralized database will inevitably leak information to the database operator; the only way to preserve full user privacy is for that user to download a local copy of the full database [14]. Moreover, centralized search engines can associate search queries with an IP address. Others can associate queries with a personal identifier by providing auxiliary, trackable services such as electronic mail. While there may still be some legal protection of this information in certain countries, in others a series of "unacceptable" searches could easily jeopardize an individual’s life, liberty, or property.

Centralized search engines must balance the interests and needs of their large user populations. However, recent work in personalization has shown that aspects of search can be tailored to the interests of an individual [32, 21, 22, 4]. Some personalizations systems are purely manual, and require that users provide a profile describing their interests to influence the ranking of results [4]. Others gather information about user intent at query time using relevance feedback [26, 9]. Recent desktop search tools even go so far as to scan the files on a user’s personal computer to compute an index that is uploaded to a central database. In contrast, the key personalization function of a microsearch engine is the personal PageRank [29], which can be computed locally and stored on the user’s personal computer. Personalization is used to influence the ranking of search results, and to select which subset of the Web should be present in the local index on the user’s microsearch engine.

Like microsearch, some search engines are intended to run near their user, although with smaller scale document repositories than the full Web. Enterprise-level search tools, such as Glimpse [27], index files and Web pages on the computers within an organization. Desktop search systems, including Google Desktop [3] and Apple’s Spotlight [7], index files and records within an individual PC’s file system.

In this paper, we have concentrated on search algorithms based on PageRank, and its personalized derivative, personal PageRank, because these are generally similar to those being used on today’s central search engines. Although better algorithms have been developed, such as Latent Semantic Indexing [16], the Semantic Web [18] and Hypertext Induced Topic Selection [25], they have so far proven too expensive for deployment on a central system servicing millions of users. With a microsearch engine, however, it is both feasible and straightforward to deploy a new search algorithm, whether to service a classic query or a new application.

Recently, Webaroo [8] released a search product that allows for disconnected Web surfing by means of a large Web “stash” [24] stored on the local disk. The system is intended for disconnected browsing, as opposed to search, and does not personalize based on user interests.

3 Prototype Implementation

In this section, we describe the architecture and implementation of Yocto, our prototype microsearch engine. Yocto provides us with a foundation for experimenting with new applications, and for measuring the performance and search quality that a microsearch engine could provide to its users today.

We built Yocto by extending the Nutch open-source search engine. Nutch provides the basic infrastructure for building a moderately scalable Web search engine; using it simplified our exploration of issues such as the utility of personalization and the selection of Web pages for inclusion in a small index. Nutch consists of about 21,000 lines of Java code. To this, we added approximately 1,900 lines of Java code and 800 lines of C code to implement our personalization functions.

Using Nutch has not been without problems. For example, the Nutch query processor is fairly rudimentary compared to many search engines, causing some queries to return poor results, reasons that are not related to microsearch itself. However, we nonetheless found Nutch to be a valuable foundation for our system.

3.1 Architectural Overview

Our Yocto prototype consists of five main software components which run on the user’s PC as shown in Figure 1. The personalizer is a client-side Web proxy that logs a user’s browsing activity. The personalizer uses these logs to determine topics and Web pages that are within the user’s interest set. Currently, the personalizer maintains access logs indefinitely and treats old and new logs with equal weight when personalizing. Over sufficiently long time-scales, it may make sense to weight recent log entries more than old log entries. As well, privacy-conscious users might want to evict information from the log after a certain period of time to avoid the risk of their browsed URLs being disclosed.

The ranker, crawler, and indexer coordinate to produce a Web page index, hyperlink graph, and a ranking over these pages that estimates their importance. The ranker computes its ranking by feeding information from the personalizer and indexer to a personalized form of the PageRank [29] algorithm, as we will describe below. The crawler downloads Web pages, preferentially retrieving pages with high rank. The indexer takes pages downloaded by the crawler, and prepares them for inclusion in the local search database. As well, the indexer extracts links from Web pages to add to a hyperlink graph, which is used by the page ranker to compute the personal PageRank.
ranks a Web page according to the likelihood that a “random surfer” would encounter that page while traversing the hyperlink structure of the Web. Conceptually, the random surfer begins on a randomly selected Web page. Next, the surfer either selects an outgoing hyperlink to follow at random, or with some fixed probability, instead “teleports” to another page from the entire Web at random. This random teleportation models the scenario in which the surfer is bored with traversing links, and instead wants to visit a new location on the Web. The PageRank for a given Web page is proportional to the probability the surfer will encounter it. In practice, PageRank is blended with other information retrieval metrics to rank search results. It exerts a strong influence on which results are returned to the user, which is the intuition why microsearch engines benefit by indexing only highly ranked pages. Pages with low PageRank are unlikely to be found in the top results for a query, so they are less useful in a small index.

Personal PageRank [29] is a variant of PageRank that is biased to the interests of a specific user. It is particularly appropriate to a microsearch engine, as its goal is to improve the coverage of a small index by ensuring that it contains pages which are likely “of interest” to the associated user. We can define Personal PageRank in terms of the same “random surfer” model of generic PageRank. The surfer wanders randomly, just as in generic PageRank, but acts differently on the occasions that he decides to “teleport”. Rather than teleporting to a random location in the Web, the surfer is heavily biased towards landing on one of a distinguished set of seed pages which reflect a user’s past surfing history. Therefore, the random walks will tend to be concentrated in the neighborhood of those distinguished pages. This concentration around the seed set helps boost the rankings of pages that are related to the user’s historical interests. Consequently, seed pages and those “nearby” will be more highly ranked.

Many different personal biasing functions are possible. In our prototype ranker, we take the simple approach of selecting teleport destination pages with probability that is proportional to the number of times that user has visited the page in the past. Pages that do not appear in the user’s log (i.e., pages the user has never visited) are assigned a much smaller, uniform probability.

3.3 Crawling

A search engine crawls the Web for two reasons. First, it crawls to uncover the hyperlink structure of the Web to facilitate computing PageRank. Second, it crawls to download the content of Web pages to construct an index. A microsearch engine could crawl the Web itself, or it could rely on the services of a centralized crawler. The former approach eliminates one more dependency on a central service and affords greater privacy, but the
resource limitations of a microsearch engine will constrain how much of the Web it can crawl.

Our prototype usually crawls the Web itself, rather than relying on or cooperating with a centralized crawler. Yocto’s crawler prioritizes what pages to crawl next according to their computed rank. To avoid impacting the user’s other network traffic, our prototype is configured to respect an overall bandwidth limit on crawling. We typically configure our prototype to retrieve approximately 250,000 pages per day, which uses only 250 Kbps network bandwidth on average. In spite of this low rate, this is more than enough to crawl all pages visited by the user during the previous day, and all pages linked to by visited pages.

Our prototype can also take advantage of a central crawler, if one is available. For example, we have a cluster of four computers that we have used to map the hyperlink structure of 313 million Web pages. Our prototype can bootstrap itself by ranking such an externally generated Web graph to select pages for its index. To investigate with the benefits of microsearch, we had Yocto identify, crawl and index slightly more than 8.5 million Web pages, consuming approximately 27 GB of storage. In the next part of this section, we describe more fully how we implemented Yocto’s index selection process.

3.4 Performance

To calibrate the performance capabilities of our prototype, we briefly describe the results of key performance microbenchmarks. The query response performance of the core Nutch search engine has been benchmarked by others [15, 13]. Summarizing their results, a Nutch installation running on a single PC is capable of serving approximately 50 queries per second out of a memory-resident index, and 2 queries per second from a disk-resident index.

Yocto’s personal PageRank calculation requires many hours to compute, depending on the size of the Web graph (Figure 2). Accordingly, we have chosen to recalculate personal PageRank once per day, during the middle of the night when it will least impact the user. The system recomputes personal PageRank taking advantage of newly crawled pages and additions to the users surfing history, after which it schedules highly ranked pages for crawling. In addition, our prototype crawls and indexes every page visited by the user during the day, along with every page linked to by those pages. Personal PageRank biases previously viewed pages towards higher rank, and hence they will rank highly in search results.

3.5 Summary

In this section, we described the design and implementation of Yocto, our prototype microsearch engine. Our implementation makes use of a personal PageRank computation to identify and download a small index; this index contains both broadly popular pages, and pages within the user’s personal interest set. In the next section of the paper, we describe a series of novel applications that we have implemented on top of Yocto.

4 New Applications Enabled by Microsearch

Although search engines have helped transform the Web, they have remained fundamentally unchanged since their introduction. As users did a decade ago, they currently interact with a search engine through a forms-based interface. From within a browser, a user submits a query, reviews the results, and jumps off to the selected page. Everyone receives the same search results, so a farmer and a computer scientist get the same results when they search for “apple.”

Microsearch enables radical changes to this traditional model of Web search. Every user can receive search results that are personalized for their unique interests, improving relevance. A microsearch engine can also be proactive, and suggest relevant information based on the user’s past or current activities. In addition, users can query their microsearch engine at times and places where a central search engine is simply inaccessible due to a lack of connectivity.

Web search can become an essential service for applications, on par with a file system or database. Applications can access both the index and the ranking information as part of their regular processing, without necessarily becoming dependent on a distant third party service provider. In this section, we describe four applications that we have implemented which are enabled by microsearch: a personalized search booster, a contempo-
raneous searcher, a personalized page recommender, and a tool for performing disconnected Web searches.

4.1 Boosting Search With Personalization

Our first application is a Personalized Search Booster, which augments generic results from a central search engine with highly personalized microsearch results. To use the booster, the user enters their query into a Web form, as usual, and receives two sets of side-by-side results. One set is from their preferred central search engine, and the other contains personalized results from their local microsearch engine. The user can review both listings and visit pages from either or both listings, or refine their search by issuing another query.

Personalization allows the microsearch engine to return surprisingly relevant results to some queries. The microsearch results are selected from the local index using the user’s personal PageRank, which is based on their Web browsing history. Anecdotally, this personal context provides the most benefit for vague or under-specified queries. The personalization helps to disambiguate and tease out what the user meant from their vague query. Table 1 shows some interesting results from one of the author’s microsearch engines, and compares them against the corresponding Google results.

4.1.1 Evaluation

To quantitatively evaluate the benefits of our search booster, we conducted a study to see if it improved upon search results provided by Google. We allocated 27GB of storage to create a 8.5 million page microsearch engine, and personalized it based on four months of surfing history provided by one of the authors. For six weeks, he used the booster for Web search and received microsearch results alongside Google results for each query. During that time, he conducted over 300 searches and clicked on results from 159 of them (he clicked no results from the remaining queries). For each query, we recorded which results he clicked on to measure the degree to which personalized microsearch provided results more relevant than Google. We tracked the outcome of each query to see whether it had been satisfied by Google’s results, the microsearch results or by selecting some results from each listing.

Figure 3 demonstrates the enhanced relevance provided by the microsearch booster. For one third of the queries, the user clicked on at least one of the microsearch results, and for 25% of queries the microsearch provided all the results he clicked on. After the study, we examined each query outcome by hand to verify the results. In the majority of cases, microsearch provided better results because of the context and disambiguation provided by personalization.

4.1.2 Discussion

There was some overlap between the results returned by the two search engines, but they each returned valuable results that were omitted by the other. When Google exclusively returned a valuable result, there are two possible reasons why it was omitted from the microsearch results. First, the microsearch index was three orders of magnitude smaller than Google’s, so it may not have even indexed that particular result. Or, the result may have been in the index but not deemed relevant to the user’s query.

Inevitably, we expect both of these deficiencies to diminish over time, and for microsearch to handle an ever growing fraction of queries. The index we used to evaluate the benefits of boosting used a modest 27GB of storage. Today’s personal computers can easily accommodate indexes an order of magnitude larger, and the cost of storage is expected to continue its recent dramatic improvements. Larger indexes can provide answers for a greater fraction of queries, an observation that we quantify in the next section. Success in sorting the right answers into the top results depends on finely-tuned relevance algorithms. Our prototype is based on the relatively immature Nutch search engine, so we expect that proper tuning would yield improved relevance with min-
Conversely, central search engines are working at a disadvantage to match the personalized relevance of a microsearch engine. Their indexes are already comprehensive, and their relevance algorithms have already undergone man-decades of tuning. The valuable microsearch results were selected as a result of personalization, and microsearch engines have sustainable advantages in this area. They have direct access to a much larger pool of personal context on the user’s personal computer, and they have dedicated resources with which to exploit this information to increase relevance.

Even if a central search engine could obtain rich personal context, it would be prohibitively expensive for it to compute a personal PageRank for each of its users. For example, a centralized system would require nearly 100,000 PCs to recompute a personal PageRank for each user just once per month, assuming a population of 10,000,000 users and a modest 160 million page Web graph. Algorithms have been proposed [22, 20] that help scale personalized ranking to large numbers of users, but all appear to sacrifice true personalization to achieve scalability. In contrast, a microsearch engine only has to compute a personal PageRank for a single user, and has sufficient resources to keep it current by recomputing daily.

Eventually, microsearch may graduate from a “boosting” role and assume a more prominent role as the user’s primary search engine. The tiny microsearch engine used in this study fully answered about 40% of the user’s queries. When storage trends and relevance tuning drive this fraction to 90% and beyond, the user may rely solely on microsearch results for most queries and only selectively probe a distant central search engine.

### 4.2 Contemporaneous Search

Our next application, **Serendipity** , performs search contemporaneously with a user’s other computing activities. Serendipity leverages the low-latency access to the microsearch engine index in order to suggest pages in real-time that are relevant to the user’s current task. Serendipity² provides a form of associative memory by constantly issuing queries based on what the user is reading or writing within an editor. As shown in Figure 4, it pulls query terms from the window containing the text which the user is viewing and displays high ranking Web pages in a small window.

The combination of a small microsearch index and aggressive, speculative querying means that many searches will not produce useful answers. From the user’s point of view, these “wasted searches” cost them nothing other than the idle local resources they consume. These costs are amortized into the benefits that accrue whenever Serendipity surprises the user with a relevant result before she even formulates a query. The dedicated local resources of a microsearch engine are able to field this heavy workload from a single user. The massive user base and economic model of centralized search engines suggest that it would be challenging for them to provide a comparable service.

### 4.3 Personalized Page Recommendation

Our next application, called **Recommender** , leverages a user’s personal PageRank information to identify pages that the user might find interesting, but has not yet discovered (Figure 5). The reading list is derived from pages with high personal PageRank that do not appear in the user’s surfing history. The user can view their microsearch-generated recommendations using any Really Simple Syndication (RSS) client. Anecdotally, the application has proven remarkably useful. For example, in the writing of this paper, **Recommender** identified two useful documents, one containing a proof about the privacy implications of centralized versus local searching [14] and another describing methods for computing

---

²Serendipity is an enhancement to an Emacs tool called Recommender [31], which performs a similar function, but only for files contained on the user’s hard drive, as opposed to Web pages.
The first widely known, crawler-based, full-text indexing Web search engine was WebCrawler [2]. At the time of publication, WebCrawler indexed nearly 50,000 documents from over 9,000 servers, handling 6,000 queries per day. Following WebCrawler, a number of increasingly large-scale search engines were commercialized and popularized, beginning with Inktomi [2] (one of first to achieve scalability using clustered computing), and including Lycos [2], AltaVista [2], and Google [2]. There are several non-commercial Web search engines, including Apache Lucene [2] and Nutch [2], on which our prototype is built.

---

## Figure 4: Contemporaneous search with Serendipity

Queries are resolved in real-time from the local microsearch engine as the user edits and reviews a document. In this example, the user configured Serendipity to help find citations, restricting search results to the CiteSeer [2] Web site.

---

### 4.4 Disconnected Search

Finally, we have built an application which allows the user to search and surf the Web during disconnected operation. If the user enables disconnected search, their microsearch engine will cache the HTML for all the pages it indexes. This approximately triples the space needed to store the index, which reduces the number of pages that can be indexed on a fixed storage budget. In exchange for these increased storage costs, the user can search and access pages when they are away from their Internet connection.

---

## 5 The Impact of Larger Disks

In the previous section we discussed the behavior of several microsearch applications running against a 27 GB index. For a computer system today, that amount of storage is not insubstantial, but for the systems of tomorrow, it will be. We are therefore motivated to understand how microsearch may perform in the future with increases in index size made possible by larger disks.

Larger disks impact a microsearch engine in two ways. First, they make it possible to index more pages, potentially increasing the relevance of a query’s results. Second, and more subtly, the greater number of pages they index can benefit from a PageRank that is computed on a larger Web graph. Ranking from a larger graph increases the accuracy of the ranking, which leads to higher quality indexes. Fundamentally, that Web graph is determined by crawling pages in order to determine their hyperlink structure.

In this section, we consider both effects in turn. In doing so, we introduce the metric of coverage to measure the likelihood that the answer to a Web query is contained within the microsearch engine’s index.

---

## 5.1 Index Size and Coverage

Large-scale, centralized search engines have vast amounts of storage capacity at their disposal. Accordingly, they can afford to index virtually the entire Web. In contrast, a microsearch engine has limited resources, and it can only afford to store an index of a subset of the
These pages have high personal PageRank for the user, who has not yet viewed them. Recommender provides a feed of these pages using the Really Simple Syndication (RSS) protocol.

Because of this, a microsearch engine’s index is at risk of missing Web pages relevant to its users’ queries. The main challenge of a microsearch engine is to select the right subset of pages to index in order to maintain high coverage. To overcome this challenge, a microsearch engine has one important advantage over its centralized counterpart: it serves only a single user, so it can use personalization to tailor which pages it indexes to best serve the needs and interests of its user. A user issues only a few queries each day, getting links to only a few hundred pages from their search engine. As long as a microsearch engine indexes those pages, its user will be satisfied.

5.1.1 Evaluation Methodology

To explore the impact of index size on coverage for a microsearch engine, we needed to gather three kinds of data: a large Web graph, a set of search queries and their relevant results, and personalization context representing the interests of a user. As previously discussed, we use the hyperlink structure of the Web graph and the personalization context to determine a ranking for every page within it. The ranking is then used to determine which pages should be represented in an index of a given size. We calculate the coverage of the index by comparing the results from a set of queries applied to the index against a “correct” result, where we define correct below.

For our Web graph, we used a modified version of the Nutch tools to crawl the Web to gather a Web graph consisting of over 313 million Web pages. We seeded our crawler with a root set of pages from the Open Directory Project as well as pages drawn from the surfing history of one of the authors.

For our search queries, we relied on two different sets of queries and their results. The first set contains 10,000 queries drawn randomly from the logs of the Excite search engine during 2001. The second set consists of 263 search queries issued by one of the authors over a period of 30 days. The first data set gives us a broad collection of queries against which to evaluate, while the second allows us to consider the impact of personalization.

We submitted all of the queries from both data sets to Google, and treated the top ranking result that it returned as “correct” for the purposes of this study. While there are certainly other ways to define correct here, none of them are testable as the Excite history contains only the queries, but no information about which results the users preferred. Consequently, we assume that Google’s ranking algorithm is a perfect predictor of correctness. As we are only looking at whether the index contains a high quality result for each query, this is a reasonable assumption.

Finally, as a simple source of personalization context, we gathered a trace of the Web surfing behavior (Table 2) of the same paper author over a period of 135 days.

Table 2: Summary statistics from tracing one of the authors. We recorded one of the author’s Web surfing history and Web searches.

<table>
<thead>
<tr>
<th>Duration of Web surfing trace</th>
<th>135 days</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of pages viewed</td>
<td>20,187</td>
</tr>
<tr>
<td>Different pages visited</td>
<td>7,796</td>
</tr>
<tr>
<td>Duration of Web search trace</td>
<td>30 days</td>
</tr>
<tr>
<td>Number of Web searches</td>
<td>263</td>
</tr>
</tbody>
</table>

5.1.2 Results

Intuitively, one would expect that coverage would increase with index size, and that a personalized ranking algorithm would have better coverage than an imperson-
alized one. The real question here though is “how much coverage can we expect from a personalized index of a given size?”

Figure 6 answers this question, by showing the coverage of indexes of various sizes populated according to PageRank and personal PageRank. To populate an index of a particular size, we simply include all the highest ranking pages and discard those of low rank. The figure shows the coverage of indexes as a function of the index size for: the 10,000 random Excite queries on PageRank indexes; and, for the author's queries using indexes built with either PageRank or Personal PageRank. For the random Excite queries, an index containing just 50 million pages is able to provide a query coverage of about 60%. In other words, a majority of the Excite queries could be resolved with a search engine having fewer than 150 gigabytes of storage. This result bodes well for a search architecture based on microsearch.

Turning to personalization, Figure 6 shows that for a given index size, personalization improves coverage, with significantly more improvement coming with smaller indexes. In practice, these differences, although small, can have a substantial impact on the required index size to achieve a given coverage because of the slopes of the coverage curves. For example, to achieve 60% coverage on the author's queries requires an index of about 27 million pages using PageRank, but only about half as many pages using Personal PageRank. Overall, our results show that an index, using storage capacity which will soon be commonplace, can satisfy a large fraction of search queries.

5.2 Crawl Size and Coverage

In our examination of coverage and index size, we made the assumption that the microsearch engine had access to a sizable Web graph by which it could calculate a personal Page Rank. Of course, in a real system, that graph must come from somewhere. As we described our prototype, we started with a precomputed graph containing substantially more pages than the index. In this subsection, we consider the effect of graph size on personal PageRank computation and index selection, as the index is increased in size. Our goal is to determine how best to obtain the graph in practice.

To evaluate the impact of crawl size, we used a small cluster of workstations to gather two different Web crawls. Our first crawl, which we call open, used as its seed set the pages described by the Open Directory Project [6]. The second, which we call personal, focused on the interests of one of the authors by using as its seed set a list of pages previously surfed by him (as described in Table 2). The open crawl ran for enough iterations to discover 243 million pages. We ran the personal crawl for fewer iterations, discovering 100 million pages. Of these, 70 million were disjoint from the open crawl. Based on these two crawls, we created a third crawl, called combined, which represents 313 million pages.

Figure 7 show that the coverage of large indexes improves when they are selected by computing PageRank on a larger Web graph. As our open crawl progressed, we computed intermediate PageRanks and used them to construct small indexes of various sizes. By using a larger Web graph to compute the personal PageRank, the coverage of an index of a given size improves. For example, a 19 million page index computed from a 100 million page Web graph obtained 50% coverage, while the same sized index computed from a 40 million page Web graph only obtained 42% coverage.

Two factors contribute to this effect. First, a smaller Web graph intrinsically contains fewer of the correct query responses, limiting the coverage an index could achieve. Second, using a smaller Web graph results in a less accurate PageRank.
Figure 7: It helps to use a large Web graph to compute PageRank in order to create smaller indexes. The coverage of an index of a given size improves if its pages are chosen from a larger Web graph. Note that we could not construct this graph using a personal PageRank. We would need to co-mingle the open and personal crawls, since the personal crawl is not large enough and the open crawl was not personalized. However, if we co-mingled them, we would not be able to identify a canonical ordering for the aggregate crawl, which we would need in order to get a range of data points along the x-axis.

5.2.1 Upper Bounds on Crawl Size

The index can only contain those pages identified during the construction of the Web graph. Consequently, we can evaluate the upper bound on coverage as a function of crawl size by assuming that the size of the index and graph are the same. To understand the impact of crawl size on coverage, we measured how many of the correct answers to the 10,000 random Excite queries had been found, as a function of the number of pages the crawler had discovered so far. Figure 8 shows our results.

The crawler was able to uncover 50% of the correct answers after crawling just 40 million pages. After crawling 243 million pages, approximately 70% of the correct answers were found. Even larger crawls would continue to find more answers, but the flattening of the curve’s slope implies that the remaining answers will require significant crawling effort to find.

5.2.2 Personalization

In Table 3, we show the number of correct results found by the open, personalized, and combined crawls, relative to the results selected from the 10K random Excite queries and the 263 author queries. Encouragingly, the personalized crawl obtained better coverage over the author’s queries than the open crawl, in spite of having fewer pages, demonstrating that personalization does indeed help. Interestingly, the coverage of the personalized crawl over the Excite query set is noticeably worse, showing that a personalized crawl benefits its user, but not the overall population.

In Figure 9, we show the number of correct pages the crawler discovered for the author’s queries, as a function of the number of pages crawled. The graph compares the open and focused crawling strategies. Both strategies quickly discovered over 60% of the results. However, by focusing on the author’s interests, the crawler discovered slightly more of the answers to the author’s queries than the open crawl, and it did so more quickly.

5.2.3 Design Implications

Overall, Figure 9 demonstrates that a microsearch engine could achieve nearly 70% coverage by crawling and indexing just 60 million pages using personalization. Once this set of pages has been identified by ranking a Web graph, a crawl of this size would consume 180 GB of storage, and would take approximately 6 days to initially populate the index, assuming a 3 Mbs broadband connection. For some applications, especially speculative ones such as Serendipity, 70% may be sufficient. Other applications though, such as a purely-local Google-like search, require higher coverage in order to produce results comparable to their centralized counterpart (see Figure 1). Crawling and storing the pages for such an index will require more resources, but crawling a
much larger Web graph for ranking purposes is the truly expensive aspect of crawling.

However, our results from the combined crawl data indicate that a cooperative crawling strategy may be the best way to achieve high coverage without requiring every PC to crawl to collect its own massive Web graph. In the cooperative approach, a centralized crawler performs a large crawl, from which it extracts the Web’s hyperlink structure. A microsearch engine then downloads this hyperlink graph from the centralized crawler and uses it to compute a personalized PageRank, allowing it to identify an effective but relatively small set of pages to crawl and include in its index. Because a hyperlink graph consumes much less storage than the actual content of the linked Web pages, a microsearch engine could afford to download a very large graph. For example, our 313 million page web graph consumes about 10GB of storage, well within the capabilities of a modern broadband-connected PC. A microsearch engine would only have to pay the cost of downloading the graph once, as the bandwidth cost of receiving incremental updates would be modest.

Regardless of the load on the local PC, as microsearch engines become more widely deployed, we expect that the cooperative approach will grow increasingly attractive since it would eliminate redundant load on Web servers and network links. In the scenario of many microsearch engines cooperating with a centralized crawler, the crawler would need to transmit large data sets to potentially many PCs. Swarming protocols like BitTorrent [1] could be used for this, as they provide cost-efficient distribution. Ideally, the provider would only need to upload a single copy of the data, which would then be redistributed through the swarm by the microsearch engines themselves.

### 5.3 Summary

In this section, we have considered the effects that larger indexes have on coverage and crawling behavior. Through trace analysis, we have shown that:

- Increasing the size of a small index can dramatically improve its coverage; larger indexes benefit from additional disk capacity, but less so.
- Personalization improves the effective capacity of an index by increasing the number of relevant entries.
- A large Web graph relative to index size improves the coverage of the index because it lends itself to a better PageRank.
- The best coverage comes from computing personal PageRank based on a Web graph that includes both broad and personally focused pages.
- Cooperative crawling is a practical strategy that offers good coverage.

### 6 Conclusions

We have made the case for a microsearch architecture which delivers Web search functionality directly on a user’s local PC. Compared to traditional centralized search, microsearch enables a new class of search-based applications, improves the relevancy of results through personalization, enhances privacy, permits disconnected operation, and allows for customized search strategies. Our prototype implementation demonstrates that a microsearch system on today’s computers makes possible new applications which deliver good results. Our analysis shows that as disk capacities increase, microsearch will be able to resolve a larger fraction of user queries, enabling it to serve an ever-expanding set of applications.

### References


