Pocket Cloudlets

Emmanouil Koukoumidis
Princeton University
Princeton, NJ, USA
ekoukoum@princeton.edu

Dimitrios Lymberopoulos
Microsoft Research
Redmond, WA, USA
dlymper@microsoft.com

Karin Strauss
Microsoft Research
Redmond, WA, USA
kstrauss@microsoft.com

Jie Liu
Microsoft Research
Redmond, WA, USA
liuj@microsoft.com

Doug Burger
Microsoft Research
Redmond, WA, USA
dburger@microsoft.com

Abstract

Cloud services accessed through mobile devices suffer from high network access latencies and are constrained by energy budgets dictated by the devices' batteries. Radio and battery technologies will improve over time, but are still expected to be the bottlenecks in future systems. Non-volatile memories (NVM), however, may continue experiencing significant and steady improvements in density for at least ten more years. In this paper, we propose to leverage the abundance in memory capacity of mobile devices to mitigate latency and energy issues when accessing cloud services.

We first analyze NVM technology scaling trends, and then propose a cloud service cache architecture that resides on the mobile device's NVM (pocket cloudlet). This architecture utilizes both individual user and community access models to maximize its hit rate, and subsequently reduce overall service latency and energy consumption.

As a showcase we present the design, implementation and evaluation of PocketSearch, a search and advertisement pocket cloudlet. We perform mobile search characterization to guide the design of PocketSearch and evaluate it with 200 million mobile queries from the search logs of `m.bing.com`. We show that PocketSearch can serve, on average, 66% of the web search queries submitted by an individual user without having to use the slow 3G link, leading to 16x service access speedup.

Finally, based on experience with PocketSearch we provide additional insight and guidelines on how future pocket cloudlets should be organized, from both an architectural and an operating system perspective.

Categories and Subject Descriptors H.4 [*Information Systems Applications*]: Miscellaneous

General Terms Design, Experimentation

Keywords Mobile Search, Mobile Cloud, Flash Storage

ASPLOS'11, March 5–11, 2011, Newport Beach, California, USA.

1. Introduction

The wide availability of internet access on mobile devices, such as phones and personal media players, has allowed users to access various cloud services while on the go. Currently, there are 54.5 million mobile internet users and market analysis shows that this number will increase to 95 million by 2013 [22], indicating that mobile devices are quickly becoming the dominant computing platform. As a result, optimizing these devices to better support access to cloud services becomes critical.

Although most cloud services have been designed to transparently support a wide range of client devices, the end user experience for these services can vary significantly across devices. Conversely to the desktop domain where the connection to any cloud service takes place over very fast and almost always available links (*i.e.*, ethernet), in the mobile domain users rely on cellular radios that tend to exhibit higher latency and unpredictability. For instance, most desktop computers can submit a search query to the search engine and successfully receive the search results in less than a second. The same task on a high-end smartphone with a 3G connection can take at least one order of magnitude more time (3 to 10 seconds depending on location, device and operator used). When the 3G radio is not connected or only Edge connectivity is available, this time can be doubled or even tripled.

Even as the throughput of radio links on mobile devices increases (*e.g.*, 4G), the user response time for the various cloud services will not be drastically improved for two reasons. First, while higher throughput links can be particularly effective for bulk transfers, recent studies [8] have shown that users tend to exchange small data packets, making link latency be the major bottleneck. Second, the radio link needs between 1.5 and 2 seconds to wake up from its standby mode even if it is already connected to the cell tower. This high startup cost is independent of the radio's throughput and is expected to hold even for future generations of radio links.

Radio links impose not only a latency bottleneck, but also a power bottleneck. Mobile devices are battery operated and the cellular radio, along with the processor and the screen, is one of the most power hungry components. The more data is exchanged and the more time the radio link is active, the lower the battery lifetime of the mobile device becomes, creating a negative user experience.

While there is such a large speed gap in accessing the internet between mobile devices and desktops, the traditional memory/storage gap between these two classes of devices is rapidly fading away. Phones and personal music players currently support up to 64GB of flash memory and future flash technologies

Figure 1. The GUI of the PocketSearch prototype. Local search results/ads are instantly displayed in the auto-suggest box as the user types the query. The GUI of the web search prototype is not shown due to space constraints.

promise much higher capacities (Section 2). The increasingly available memory resources on these devices can transform the way mobile cloud services are structured. As more and more information can be stored on mobile devices, specific parts of or even full cloud services could be transferred to the actual mobile devices, transforming them into *pocket cloudlets*.

Pocket cloudlets could drastically improve the mobile user experience in three major ways. First, since all or portions of the information resides on the phone, users can instantly get access to the information they are looking for, eliminating, when possible, the latency and power bottleneck introduced by the cellular radio. Furthermore, by serving user requests on the actual device, pocket cloudlets can mitigate pressure on cellular networks, which is expected to be a critical resource as mobile internet grows. Second, since most of the interactions between the user and the service take place on the mobile device, it is easier to personalize the service according to the behavior and usage patterns of individual users. Third, since the service resides on the phone, all the personalization information could also be stored on the phone and possibly protect the privacy of individual users.

For instance, in the case of a search engine, a large part of the web index, ad index and local business index can be stored on the phone enabling users to instantly access search results locally without having to use the cloud. In essence, a mini search engine could be running on the phone providing real-time search results to the mobile user. In another setting, the actual web content could also be cached on the mobile device to provide an instant browsing experience. Web content that might be of interest to the user could be automatically downloaded to the user's phone overnight and become available during the course of the day.

Independently of what cloud service is replicated on the mobile device, pocket cloudlets enable a fast and personalized mobile user experience. In this paper we describe a new architecture of mobile cloud services that takes advantage of the increasing non-volatile memory sizes to alleviate the latency and power bottlenecks introduced by cellular radios on mobile devices. First, we show an analysis on how NVM capacity is expected to increase in the future. Then we propose the pocket cloudlet architecture that leverages both community and user behavior to provide an instant mobile user experience reducing, when possible, overall service latency and energy consumption.

We then present in detail the design, implementation and evaluation of PocketSearch, a pocket cloudlet that replicates a search and advertisement engine on an actual phone. We analyze 200 million queries to understand how mobile users search on their phones, and then utilize the results of this analysis to guide the design of PocketSearch. Using a prototype implementation (Figure 1) and real search query streams extracted from mobile search logs of `m.bing.com`, we show that PocketSearch is able to successfully

	Flash				Other NVM technology				
year	'10	'12	'14	'16	'18	'20	'22	'24	'26
tech (nm)	32	22	16	11	11	8	5	5	5
scaling factor	1	2	4	8	8	16	32	32	32
chip stack	4	4	6	6	8	8	12	12	16
cell layers	1	1	1	2	2	4	4	8	8
bits per cell	2	3	2	2	2	1	1	1	1

Table 1. Technology scaling trends.

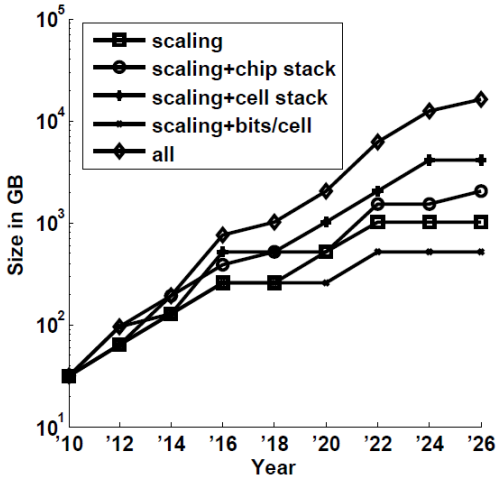


Figure 2. Memory size evolution for high-end smartphones assuming the trends shown in Table 1.

serve, on average, 66% of all web search queries submitted by an individual user. The benefits are twofold. From the user perspective,two thirds of the queries can be answered within 400ms, which is 16 times faster when compared to querying through the 3G link. From the search engine perspective, two thirds of the query load can be eliminated resulting in significant cost savings and easier query load balancing during peak times.

2. Mobile NVM Scaling Trends

When it comes to pocket cloudlets, the first question we need to answer is: "what capacities of NVMs are likely to be available in future mobile devices?" A closer look at technology scaling trends helps understand how the capacity of NVM memories in mobile devices is expected to evolve and get so abundant that it fundamentally changes the balance of cloud service implementation and makes the concept of proactively pushing large amounts of information into mobile devices appealing.

Table 1 shows our somewhat *conservative* technology scaling projections in the NVM market over the next 16 years. We assume that flash will dominate this market until it runs into charge-based storage scaling issues in the 2016/2018 time frame. At this point, we assume it will be replaced with another NVM technology more resilient to smaller feature sizes, such as resistive (*e.g.*, PCM [5], RRAM [14]) or magneto-resistive memories (*e.g.*, STT-MRAM [12]).

The first data row of Table 1 shows a projection of how the number of cells per layer in NVM memory devices is expected to scale in the form of a scaling factor. During the period in which flash is used as the NVM technology of choice, it is projected to double in capacity every two years [9].

In 2018, flash may lose traction due to increasing challenges in storing state (*i.e.*, electrons) in charge-based cells. A new technology providing more stable cells at smaller features may at that point replace flash. Single-layer PCM is already being productized

Pocket Cloudlet	Single Item	Number of Items
Web Search	100 KB (search result page)	$\approx$ 270,000
Mobile Ads	5 KB (ad banner)	$\approx$ 5,500,000
Yellow Business	5 KB (map tile with business info)	$\approx$ 5,500,000
Web Content	1.5 MB (`www.cnn.com`)	$\approx$ 17,500
Mapping	5 KB (128x128 pixels map tile)	$\approx$ 5,500,000

Table 2. Number of data items that can be stored in 25.6GB (10% of the projected NVM size available on low-end smartphones) for different pocket cloudlets.

as replacement for NOR flash in mobile devices, so it is a good candidate. The shift from flash to another technology could cause significant disruption in fabrication processes and would likely cause scaling to stall for one generation. Scaling is likely to resume in subsequent years until it finally stops in 2022 when industry is expected to hit $5nm$ technology. The second row shows chip stacking projections. Two layers are added every four years until 2022, when the technology may be mature enough to start making increments of four layers every four years. The third row shows a progression of number of layers when cell stacking is employed. Cell stacking is a technique by which devices are fabricated in multiple layers on the same silicon base (instead of in independently fabricated chips that are then combined, as in chip stacking) [6]. The process of taking a technology from academic demostration to initial production is typically five years. Given that this technology was demonstrated in 2009, this technology is likely to be adopted circa 2016 and the number of layers is likely to double every four years. Finally, the fourth row shows number of bits per memory cell. This number should increase in the next few years for flash, but then start to decrease as feature sizes get smaller, process variation increases and the average number of electrons per cell drops. In such a setting, even small electron losses may cause cell state to be corrupted, which in turn forces designers to reduce the number of logic levels, and therefore bits per cell, to increase the distance between these levels and better distinguish stored values.

Assuming the trends described above, Figure 2 presents various evolution scenarios for NVM parts used in smartphones. Our projections start with the NVM storage found in a high-end smartphone in 2010. We then apply different combinations of scaling and other capacity-increasing techniques to make a projection of total NVM capacity in future smartphones. Figure 2 shows that high-end phones may reach 1 TB of NVM as early as 2018. Considering that low-end smartphones today have 512 MB of NVM, a ratio of 64-to-1 when compared to high-end smartphones, we can calculate that low-end phones may eventually reach 256 GB (16 GB in 2018), still a respectable amount of storage.

Dedicating only 10% of a 256 GB NVM memory to caching services results in 25.6 GB of storage available in a mobile device. This storage could be used by various cloud services to offer an instant mobile user experience through pocket cloudlets. Table 2 shows the number of data items (*i.e.*, search result pages, web sites, etc.) that can be stored in 25.6 GB of space for various pocket cloudlets. A low-end smartphone is projected to be able to store more than 5 million map tiles or 17000 web sites. To put these numbers in perspective, our search log analysis indicated that more than 90% of mobile users visit fewer than 1000 URLs over a period of several months, which is 17 times fewer than the number of web sites that we can actually store on the phone. For the mapping service, assuming that each map tile covers 300x300 meters of actual earth surface, 5.5 million map tiles can cover the area of a whole state in the United States. As a result, the projected memory resources for smartphones could easily sustain the web browsing and mapping needs of a typical mobile user.

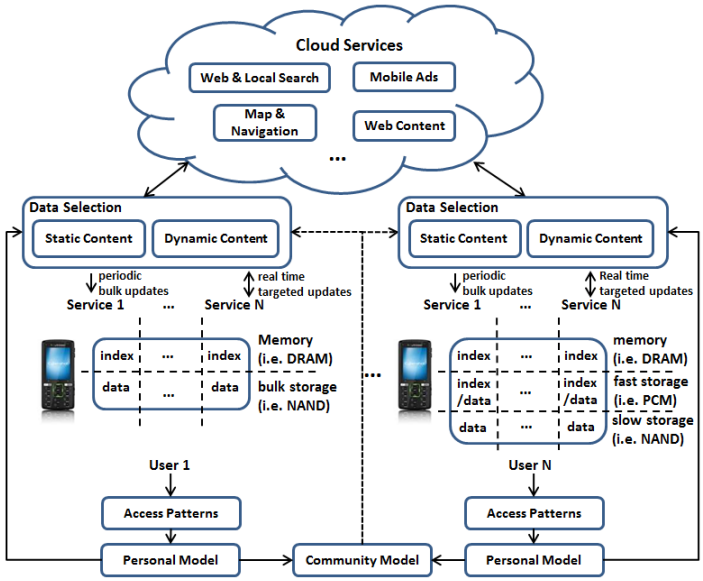


Figure 3. The envisioned architectural support required to transform mobile devices into pocket cloudlets.

3. Pocket Cloudlet Architecture

Enabling mobile devices to efficiently host cloud services poses various challenges. First, the amount of data to be stored locally on the device needs to be determined for each cloud service. Even though the analysis in Section 2 shows that memory resources on mobile devices will be abundant in the next decade, the amount of data available on the internet and across the different cloud services may exceed the available memory resources on a typical smartphone. Second, a mechanism to manage the locally stored cloud data is required as this data might change over time (*e.g.*, web content changes over time). Third, a storage architecture for efficiently storing and accessing this large amount of data is needed. Mobile users need to be able to quickly search and access data across services while still having enough space to store their personal data. Figure 3 shows the infrastructure required to transform mobile devices into pocket cloudlets.

3.1 Data Selection

At a higher level, the data stored locally on the mobile device is selected based on both personal and community access models. The access patterns of the individual user to a specific service (*i.e.*, web content or web search) are recorded and used to construct a personal model (*e.g.*, favorite web pages or search topics). At the same time, the personal models across users are combined together into a community model that identifies the most popular parts of the cloud service data across all users. Both personal and community models are then used to identify the most frequently accessed parts of the cloud service data that is or might be of interest to the individual user.

3.2 Data Management

The locally cached cloud service data needs to get updated periodically (*e.g.*, nightly, weekly or monthly). Updates occur when the device has access to power resources and high bandwidth links (*i.e.*, charging and connected to WiFi or tethered to a desktop computer). Periodic updates based on the charging state of the device are appealing, but can only be effective for relatively static data. That is, data that is not very frequently updated. For instance, the search index or the map tiles used for search and mapping services

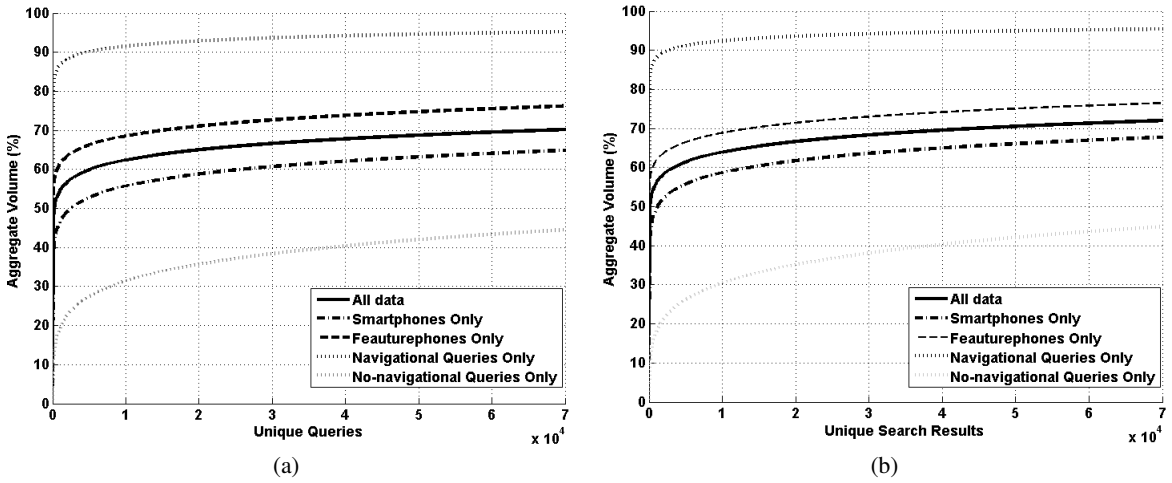


Figure 4. CDF plots of the (a) query volume and (b) clicked search result volume.

are examples of static data that could be updated periodically and only when the mobile device is charging without hurting the quality of the cloud service. However, not all cached data tend to be static. For instance, web content such as news and stock prices is dynamic in nature and tends to be accessed by individual users several times within a day. For this type of pocket cloudlets, real time updates over the radio link are required to guarantee freshness of the cached data. However, performing bulk updates over power hungry and bandwidth limited radio links is inefficient, if not impossible. Luckily, it turns out that the amount of dynamic data that is repeatedly accessed by mobile users tends to be small. For example, the analysis of 200 million mobile search queries submitted by hundreds of thousands of users showed that 70% of web visits tend to be revisits to less than a couple of tens of web pages for more than 50% of the users. As a result, instead of enforcing inefficient bulk updates over the radio link, only the small set of most frequently visited data (identified by the access patterns of the individual user) is updated in real time.

3.3 Architectural Implications

At a lower level, each cloud service owns its own storage space on the mobile device and uses it to store the necessary data. For instance, the storage can mirror web pages in the case of the web content service, and map tile snapshots and local business information in the case of mapping and navigation services. Since the amount of data required by these services is expected to be large and should always be available on the device even after a power down, bulk non-volatile storage such as NAND flash is a suitable memory technology. Besides storing the actual data on the mobile device's NVM, each cloud service also maintains an index of its data in fast volatile memory (DRAM). The index enables instant retrieval of the required data from bulk storage. Given the characteristics of current memory technologies, the main memory of the phone (*i.e.*, DRAM) is able to provide the necessary performance and density for storing the different indexes.

However, as new memory technologies such as PCM mature, this two-tier memory structure might slowly evolve into a three-tier structure as shown in Figure 3. PCM memory could become the intermediate tier by filling the performance and storage density gap between DRAM and NAND flash. In practice, PCM could be seen as fast bulk storage when compared to NAND flash, making it an ideal technology for storing the data indexes. While slower than DRAM, PCM has the advantage of being non-volatile and significantly faster than NAND flash. Being able to store data indexes in PCM eliminates the need to commit to and load the index from NAND after each power cycle of the mobile device. Given that the data required by the cloud services might be in the order of tens or even hundreds of gigabytes, the size of the data indexes can reach gigabytes, making its transfer between flash and main memory extremely time consuming. By introducing a PCM-based layer, all data indexes could become instantly available on the device at boot time offering a much faster user experience. At the same time, the DRAM tier could be used to cache the most frequently accessed parts of the data indexes in order to provide the fastest user experience when possible.

Figure 3 only shows the high level architectural requirements to enable mobile cached cloud services. In practice, however, several lower level challenges and design tradeoffs can arise. In the next sections, we present these challenges and tradeoffs for an example cloud service that focuses on mobile search and advertisement.

4. Cacheability of Mobile Search

Before designing and implementing a pocket cloudlet for a cloud service, it is important to study its cacheability. For instance, the real impact of a search pocket cloudlet depends on the fraction of the query volume that can be successfully served locally on the device. To answer this question we analyzed 200 million queries, submitted to `m.bing.com` over a period of several consecutive months in 2009. The query volume consisted of web search queries submitted from mobile devices such as phones and personal music players. Every entry in the search logs we analyze contains the raw query string that was submitted by the mobile user as well as the search result that was selected as a result of the submitted query. No personal information, such as location, is included in the logs.

4.1 The Mobile Community Effect

First, we examine the community of mobile users as a whole to discover caching opportunities across users. From the web search logs, we extract the most popular queries submitted and the most popular search results clicked by the mobile users. Figures 4(a) and 4(b) show the cumulative query and search result volume as a function of the number of most popular queries and search results respectively. When looking across all data, it turns out that the 6000 most popular queries and the 4000 most popular search results are responsible for approximately 60% of the query and search result volumes respectively. In other words, there is a small set of queries and search results that is popular across all mobile users. This suggests that if we store these 6000 queries and 4000 search results locally on the phone we could theoretically answer 60% of

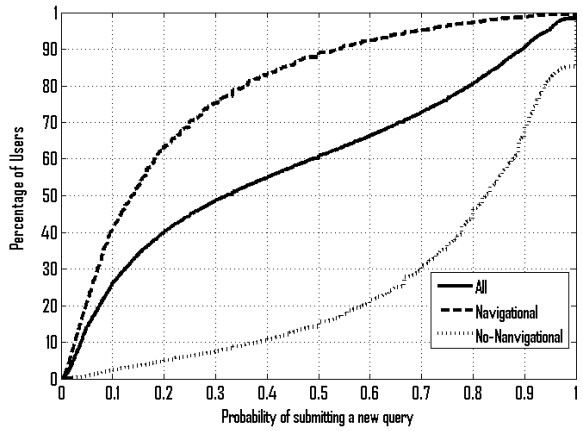


Figure 5. CDF plot of the repeatability of mobile search queries across individual users over a 1-month period.

the overall queries submitted by mobile users without having to use the radio link.

Note that these 4000 search results might not always point to static web pages. However, while some web content might be highly dynamic, the search results and queries that point to it can be relatively static. For instance, the CNN web page (`www.cnn.com`) is updated every minute and sometimes even more frequently. However, the way mobile users reach this dynamic web page is relatively static (*e.g.*, search for "cnn" or "news" and then click on the static search result that points to the CNN web page).

Similar popularity trends exist for desktop queries. However, mobile queries are significantly more concentrated than desktop queries. For instance, the first 6000 queries represent 60% of the query volume in the mobile domain but less than 20% of the query volume in the desktop domain [19].

We further divide the queries in two different categories, navigational[1] (*i.e.*, "youtube" or "facebook") and non-navigational (*i.e.*, "michael jackson"), and we study the same trends for each category. As Figure 4 shows, both query types follow the same trends but navigational queries are significantly more concentrated compared to non-navigational queries. For instance, the first 5000 navigational queries are responsible for 90% of the navigational query volume while the same number of non-navigational queries accounts for less than 30% of the non-navigational query volume.

Another interesting observation comes from comparing the results between Figures 4(a) and 4(b). To achieve a cumulative volume of 60%, 50% more queries are required compared to the number of search results (6000 queries vs 4000 search results). The search logs show that users search for the same web page in many different ways. For instance, mobile users often either misspell their queries because of the small keyboards they have to interact with (*i.e.*, "yotube" instead of "youtube") or purposely change the query term to reduce typed characters (*i.e.*, "boa" instead of "bank of america"). However, even though a misspelled or altered query is submitted, the search engine successfully provides the correct search result and thus a successful click through is recorded. As a result, a popular webpage is, in general, reached through multiple search queries.

Figures 4(a) and 4(b) also show the same information when considering the queries and search results that were submitted by featurephone (low-end mobile devices with limited browsers and internet capabilities) and smartphone users in isolation. Even

[1] A query is classified as navigational when the actual query string is a substring of the clicked URL (*i.e.*, "youtube" and `www.youtube.com`)

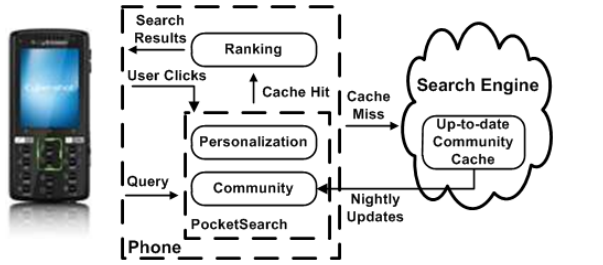


Figure 6. Overview of the PocketSearch cloudlet.

though the exact same observations hold in both cases, queries and search results that are accessed over feature-phones are in general more concentrated when compared to smartphones. This is an artifact of the limited user interfaces found on feature-phones that make web access a challenging task.

4.2 The Individual Mobile User Effect

Next, we examine personal query traces to discover caching opportunities within the search habits of individual users. In particular, we study how often individual users repeat queries. We call a query a repeated query only if the user submits the same query and clicks on the exact same search result. Figure 5 shows the percentage of individual mobile users as a function of the probability of submitting a new query within a month. Approximately 50% of mobile users will submit a new query at most 30% of the time. Thus, at least 70% of the queries submitted by half of the mobile users are repeated queries. Figure 5 also shows this trend for both navigational and non-navigational queries.

Consequently, knowing what the user searched in the past provides a very good indication of what the user will search for in the future. Again, this trend of repeating queries is not unique to mobile queries. Desktop users also tend to repeat queries, but not as frequently as mobile users. Recent studies have shown that desktop users will repeat queries on average 40% of the time [28] as opposed to 56.5% for mobile users (Section 4.2.1).

5. PocketSearch Architecture

Given the mobile search trends that the analysis of the web search logs highlighted, we designed and implemented PocketSearch, a mobile search pocket cloudlet that lives on the phone and is able to answer queries locally without having to use the 3G link[2]. The goal of the PocketSearch architecture is to capture the locality of mobile search as demonstrated in Section 4, and to be computationally tractable so that it can efficiently run on a mobile device. PocketSearch accomplishes both of these goals making its underlying architecture a template for other pocket cloudlet services.

Note that PocketSearch is not aiming to replace the actual search engine. Instead, its goal is to assist the search engine to provide a much faster mobile search experience in a way that is transparent to the user. For instance, most of the high-end smartphones today can automatically provide query suggestions to the user almost instantly and as the user is typing his query. PocketSearch's ability to retrieve search results fast, can make this experience richer by enabling the display of actual search results along with auto-suggest query terms in the auto-suggest box in real time (Figure 1). If users are not interested in any of these search results, they can access the latest set of search results through the 3G radio by

[2] PocketSearch only stores search results and not the actual web content that these results point to. Another cloudlet responsible for web content caching/pre-fetching (*i.e.*, PocketWeb) running along with PocketSearch could be used to serve the actual web content.

Query	Search Result	Volume
michael jackson	www.imdb.com/name/nm0001391/bio	10^6
movies	www.fandango.com	$95 * 10^4$
michael jackson	www.azlyrics.com/j/jackson.html	$90 * 10^4$
ringtones	www.myxer.com	$50 * 10^4$
pof	www.plentyoffish.com	$20 * 10^4$
...	...	...
Total Volume		$50 * 10^5$

Table 3. A list of query-search result pairs sorted by their volume is generated by processing the mobile web search logs over a time window (*i.e.*, a month). The volume numbers used in this table are hypothetical.

selecting the query term provided by auto-suggest or by entering the full query term on their own.

The PocketSearch cloudlet can be used to store web search results, local businesses as well as mobile advertisements (Figure 1). However, in the interest of space and to describe the underlying architecture in greater clarity, we limit this paper into describing the proposed caching system in the context of web search caching.

PocketSearch consists of two discrete but strongly interrelated components; the community and the personalization components (Figure 6). The community part of the cache is responsible for storing the small set of queries and search results that are popular across all mobile users. This information is automatically extracted from the search logs and is updated overnight every time the mobile device is recharging, making sure that the latest popular information is available on the mobile device. The community part serves as a warm start for the cache and enables PocketSearch to instantly provide search results without requiring any previous knowledge of the user.

The personalization part of the cache monitors the queries entered as well as the search results clicked by the user and performs two discrete tasks. First, it expands the cache to include all those queries and search results accessed by the user that did not initially exist in the community part of the cache. In that way, the cache can take advantage of the repeatability of the queries submitted by the mobile users to serve as many queries as possible locally on the mobile device. Second, it collects information about user clicks, such as when and how many times the user clicks on a search result after a query is submitted, to customize ranking of search results to user's click history.

When a query is submitted, PocketSearch will first perform a lookup in the cache to find out if there are locally available search results for the given query. In the case of a cache hit, the search results are fetched from the local storage, ranked based on the past user access patterns recorded by the personalization part of the cache, and immediately displayed to the user. In the case of a cache miss, the query is submitted to the search engine over the 3G radio link.

Realizing the architecture shown in Figure 6 on an actual mobile device poses several challenges:

Content Generation: A methodology is required to decide which and how many queries and search results should be included in the cache.

Storage Architecture: An efficient way to store and quickly retrieve the search results on the mobile device is needed. Memory overhead should be minimized to prevent performance degradation on the device and provide ample storage space for user's personal files. At the same time, PocketSearch should be able to quickly locate and retrieve the search results to minimize user response time.

Personalized Ranking: The user's search patterns provide important information about the individual user's interests. PocketSearch should record and leverage this information over time to personalize the search experience.

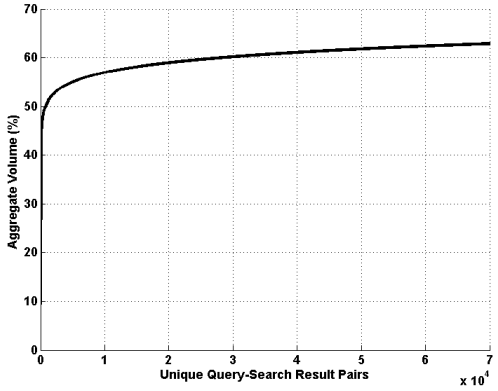


Figure 7. Cumulative query-search result volume as a function of the most popular query-search result pairs.

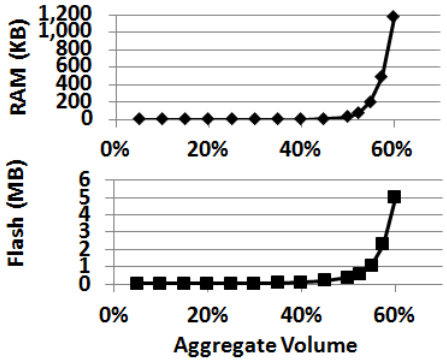


Figure 8. PocketSearch's DRAM and flash overhead for different *query-search result* aggregate volumes.

Cache Management: A scalable mechanism for regularly updating the cache contents is required. Having available the most up-to-date set of popular queries and search results on the phone is necessary for maximizing the number of queries that can be served by PocketSearch.

5.1 Cache Content Generation

The search results stored in the cache are extracted directly from the mobile search logs. The goal of this process is to identify the most popular queries and search results that are of interest to the mobile community.

A set of triplets in the form <query, search result, volume> are extracted from the search logs and sorted based on *volume* (Table 3). The term *query* corresponds to the query string submitted to the search engine, the term *search result* corresponds to the search result that was selected after entering the query, and the term *volume* represents the number of times in the search logs that the specific *search result* was selected after entering the query string *query*. For instance, the first row in Table 3 can be interpreted as follows: *In the last month, there were 1 million searches where the search result* `www.imdb.com/name/nm0001391/bio` *was selected after the query "michael jackson" was submitted.*

The number of triplets in Table 3 can be in the order of tens or hundreds of millions. Storing all of them would require significant memory resources that a phone might not be able to provide or a user might not be willing to sacrifice.

Deciding which entries to store is straightforward. To maximize the query volume that can be served by the cache, we should always store the most popular pairs of queries and search results indicated by the top entries of Table 3.

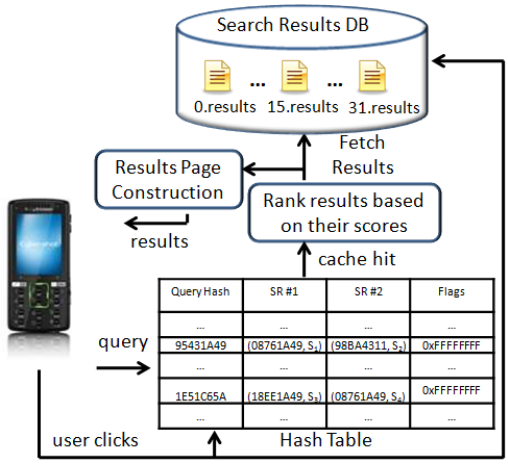


Figure 9. Overview of PocketSearch's storage architecture.

Deciding how many of the most popular *query-search result* pairs to store is a more complicated process. We select the number of *query-search result* pairs to cache based on either a memory or cache saturation threshold.

Memory (NAND flash or DRAM) Threshold: Starting from the top entry in Table 3, we run down through its entries and continuously add *query-search result* pairs until a specific flash or DRAM memory size threshold M_{th} is reached. This threshold can be set by either the phone itself based on its available memory resources or by the user, depending on how much storage space and memory the user is willing to sacrifice for PocketSearch.

Cache Saturation Threshold: Starting from the top entry in Table 3, we run down through its entries and continuously add *query-search result* pairs until we reach a *query-search result* pair with a normalized volume lower than a predetermined threshold V_{th}. The normalized volume of a *query-search result* pair is generated by dividing this pair's volume by the total volume of all *query-search result* pairs in the search logs. For instance, the *normalized volume* of the first query-search result pair in Table 3 is equal to: $10^6/5 * 10^6 = 0.2$.

The value of the cache saturation threshold is illustrated in Figure 7. It is apparent that the value of adding *query-search result* pairs quickly diminishes. In particular, slightly increasing the aggregate volume from 58% to 62% requires to double the amount of *query-search result* pairs from 20000 to 40000.

In practice, the mobile web search log analysis we performed showed that the cache saturation threshold will be quickly reached before PocketSearch stretches the memory or storage resources available on the phone. This can be seen in Figure 8 that shows the size of DRAM and flash required by PocketSearch as a function of the aggregate *query-search result* volume represented by all the pairs stored in the cache. It is clear that the saturation point of the cache is quickly reached when the most popular *query-search result* pairs that correspond to approximately 55% of the cumulative *query-search result* volume has been cached. At this point, the cache requires approximately 1MB of flash and 200KB of DRAM, which accounts for less than 1% of the available memory and storage resources on a typical smartphone.

Independently of which threshold is used (memory or cache saturation), this methodology identifies the n top entries in Table 3. Each of these n top *query-search result* pairs is then associated with a ranking score that is produced by normalizing its volume across all search results that correspond to the query. For instance, in the case of query "michael jackson" in Table 3, the ranking score for the $imdb$ search result is equal to $10^6/1.9 * 10^6 = 0.53$ and the

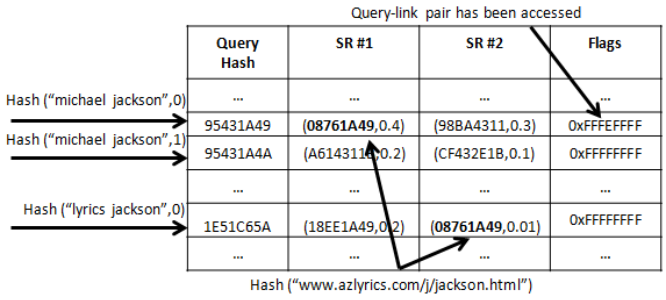


Figure 10. The hash table data structure used to link queries to search results.

score for the $azlyrics$ is $9 * 10^5/1.9 * 10^6 = 0.47$. The generated <query,search result,score> triplets can now be used to build the cache on the phone.

Extracting PocketSearch's cache contents directly from the mobile search logs provides several advantages. First, even though there might be tens or even hundreds of search results available for a given query, we only cache these search results that are popular across all mobile users, limiting the amount of memory resources required. Second, each query and search result pair extracted from the search logs is associated to a ranking score, enabling the phone to rank search results locally. Third, by processing the mobile search logs we automatically discover the most common misspellings and shortcuts of popular queries, enabling PocketSearch to cache search results for these cases. As a result, queries such as "pof", and "boa" can now be served locally on the phone by instantly displaying search results such as `www.plentyoffish.com` and `www.bankofamerica.com` respectively.

5.2 Storage Architecture

The extracted set of <query,search result,score> triplets must be efficiently stored on the phone. Storage efficiency is defined in two ways. First, the memory resources required to store the search results should be as low as possible to permit many pocket cloudlet services to concurrently run. Second, the time it takes to retrieve, rank and display search results after the user enters a query should be as low as possible.

Figure 9 provides an overview of PocketSearch's storage architecture. It consists of two components, a hash table and a custom database of search results. The hash table lives in main memory and its role is to link queries to search results. Given a query the hash table can quickly identify if we have a cache hit or a cache miss. In the case of a cache hit, the hash table provides pointers to the database where the search results for the submitted query are located. Along with each search result pointer, the hash table provides its ranking score, enabling PocketSearch to properly rank search results on the phone.

The custom database of search results resides in flash and its role is to store all the available search results so that they occupy the least possible space and they can be quickly retrieved. The data stored in the database for each search result includes all the necessary information for generating the same search user experience with the search engine: the actual web address, a short description of the website and the human readable form of the web address.

Over time and as the user submits queries and clicks on search results, PocketSearch updates both the hash table and the database of search results. Every time the user clicks on a search result, its ranking score is properly updated in the hash table. In addition, if a new query or a new search result is selected that does not exist in the cache, both the hash table and the database are properly updated so that this query and search result can be retrieved from the cache in the future.

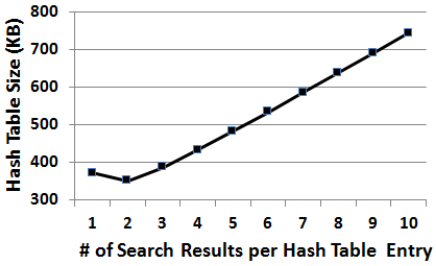


Figure 11. The memory footprint of the hash table for different number of search results per hash table entry.

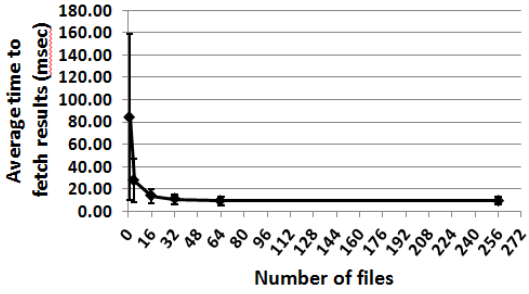


Figure 12. Average time to retrieve two search results from the database as a function of the number of files used to store the search results. The vertical bars represent the deviation of the access time over 10 consecutive experiments.

5.2.1 Query Hash Table

Figure 10 shows the structure of the hash table used to link queries to search results. Every entry in the hash table corresponds to one and only one query and has 4 fields. The first field contains the hash value of the query string that this entry corresponds to. The next two fields are of identical type and represent two search results associated to the query (SR #1 and SR #2 in Figure 10.). As it is explained later in detail, only two search results are stored per entry to minimize hash table's memory footprint. Each search result in the hash table is represented by a pair of numbers. The first number corresponds to the hash value of the web address of the search result. This value is used to uniquely identify a search result and, as it will be described in the next section, is used as a pointer to retrieve the information associated to the search result (short description, web address etc.) from the database. The second number corresponds to the ranking score of the search result. The last field of each entry in the hash table is a 64-bit number that is used to log information about the two search results in this entry. Currently, we use only one bit for each search result to indicate if the user has ever accessed the specific *query-search result* pair. The rest of the flag bits are reserved for future purposes.

In general, given a set of <query,search result,score> triplets the hash table is generated as follows. For every unique query in the set of triplets we identify all the search results associated to this query. An entry is created in the hash table for the query and search results are added in descending order of score. If more than two search results are associated to the same query, additional entries are created in the hash table by properly setting the second argument of the hash function (*i.e.*, "michael jackson" query in Figure 10).

This approach of linking queries to search results highlights two important design decisions that were influenced by the properties of the <query,search result,score> triplets extracted from the mobile web search logs. First, the number of search results linked to a query in a hash table entry affects the memory footprint of the hash

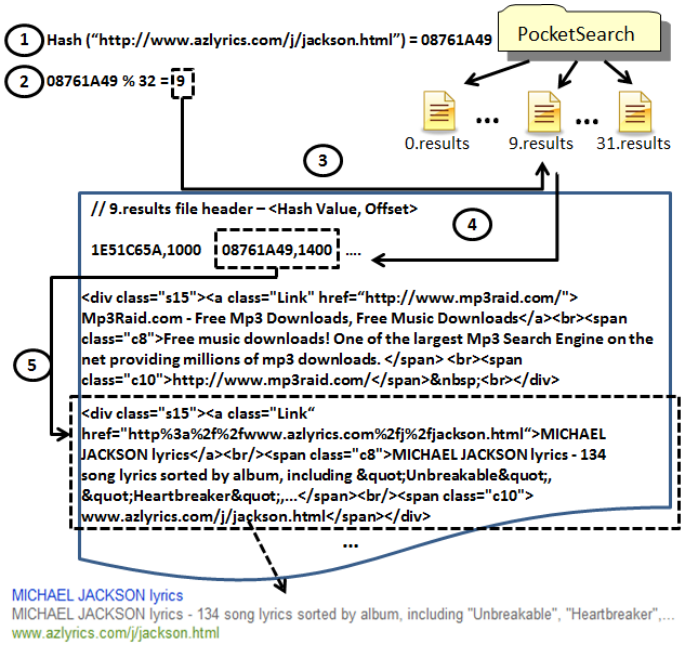


Figure 13. The 32-file custom database and illustration of the process of retrieving a search result.

table. This is illustrated in Figure 11 that shows the memory footprint of the hash table for different numbers of search results stored per hash table entry. The smallest memory footprint is achieved when two search results are stored per hash table entry.

Second, the way queries are linked to search results can affect the storage requirements of the database of search results. The simplest and fastest approach to retrieving and displaying search results to the user would be to store them in a single HTML file. Even though this approach would simplify the structure of the hash table, it would significantly increase the flash memory required to store them. The reason is that most of the search results are shared across a large number of queries. The analysis of the logs indicates that only 60% of the search results in PocketSearch are unique. If a single search result page were to be stored for every query, then 40% of the search results would have to be stored at least twice. To avoid wasting flash resources, we opted to store each search result once and then link individual queries to each search result independently. Besides saving space, this approach also enables PocketSearch to easily add/remove search results to/from the hash table and update the ranking score of search results over time. As it will be shown in Section 6, the overhead introduced by this approach in terms of user response time is negligible.

5.2.2 Search Results Organization

Search results are stored in flash using a custom database of plain text files to ensure portability of PocketSearch across different mobile platforms. For every search result, we store its title, which serves as the hyperlink to the landing page, a short description of the landing page and the human readable form of the hyperlink (Figure 13).

The amount of memory required to store the information associated to a search result into a file is, on average, 500 bytes. However, the actual memory space required might be significantly higher due to the internal structure of flash chips. Flash memories are organized in blocks of fixed size that are usually equal to 2KB, 4KB or 8KB depending on the size of the chip. If we store a 500 bytes file containing a single search result in flash memory, then this file

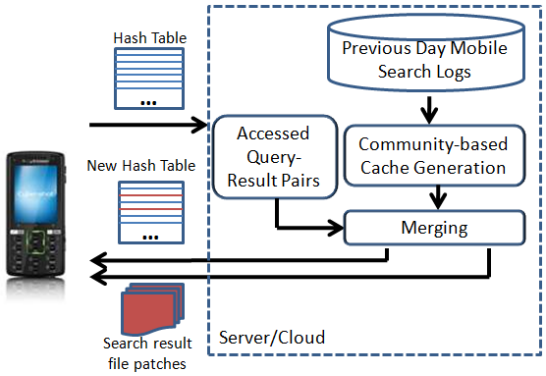


Figure 14. Overview of PocketSearch's updating process.

will occupy 4, 8 or 16 times more flash space than its actual size depending on the block size that is used.

In order to avoid flash fragmentation, multiple search results should be aggregated and stored into as few files as possible. However, storing a large number of search results into a single file could increase the time it takes PocketSearch to locate and retrieve a search result and, thus it could hurt the response time of the cache. As a result, the way search results are aggregated into files and organized within a file is critical for minimizing both, flash fragmentation and cache response time. By evaluating different database organizations (Figure 12) we found that a number of 32 database files constitutes the best tradeoff between flash fragmentation and user response time.

Figure 13 shows how search results are organized within a database file when 32 database files are used. Each search result is assigned to one of the 32 files based on the hash value of its web address. In particular, the remainder of the division of the hash value with the number of files in the database (a number between 0 and 31) is used to identify the file where the search result should be stored.

The first line in each of the 32 database files contains pairs of the form $(hash\,value,\ offset)$. The *offset* represents the actual offset from the beginning of the file where the information for the search result represented by the *hash value* is located. By parsing the first line of a database file we can identify where each search result stored in this file is located. Whenever the user clicks on a search result that is not already cached, PocketSearch will add the search result at the end of the database file and augment the header of this file with the $(hash\,value,\ offset)$ pair for this search result.

5.3 Personalized Ranking

By monitoring user clicks over time, the personalization component of the cache is aware of when and how many times the user selects a search result after a given query is submitted. PocketSearch uses this information to incrementally update the ranking score of the cached search results to offer a personalized search experience.

Assume that for a query Q there are two search results R_1 and R_2 available in the cache. Every time the user submits the query Q and clicks on the search result R_1, PocketSearch updates the scores S_1 and S_2 for the two search results R_1 and R_2 respectively, as follows:

$$S_1 = S_1 + 1 \quad (1)$$

$$S_2 = S_2 * e^{-\lambda} \quad (2)$$

The ranking score of the selected search result is increased by 1 (Equation (1)), the maximum possible score of a search result extracted from the mobile search logs. In that way, we always favor search results that the user has selected. Note that if this search result did not initially exist in the cache (selected after a cache miss), then a new entry in the hash table is created that links the submitted query to the selected search result and its score becomes equal to 1. At the same time, the ranking score for the unselected search result is exponentially decreased [3] (Equation (2)). This enables PocketSearch to take into account the freshness of user clicks. For instance, if search result R_1 was clicked 100 times one month ago and search result R_2 was clicked 100 times during last week, then the ranking score for R_2 will be higher.

Using Equations (1) and (2), the ranking score of the search results, at any given time, reflects both the number and freshness of past user clicks. In practice, any personalization ranking algorithm [27],[28] could be used with the proposed cache.

5.4 Cache Management

Figure 14 provides an overview of the mechanism used to update the community component of the cache. The phone transmits to the server its current version of the hash table. The server runs through the hash table and removes all the *query-search result* pairs that have not been accessed by the user in the past. This can be easily done by examining the *flags* column in the hash table (Figure 10). The *query-search result* pairs that have been accessed by the user in the past are only removed from the cache when their ranking score becomes lower than a predetermined threshold (*i.e.*, the user hasn't accessed the search result over the last 3 months).

At the same time, the server periodically (*i.e.*, daily) extracts the most popular queries and search results from the mobile search logs as described in Section 5.1, and adds them to the hash table. During this process, conflicts might arise in the sense that a *query-search result* pair that already exists in the hash table (previously accessed by the user) might re-appear in the popular set of queries and search results extracted on the server. The conflict is caused when the ranking score stored in the hash table is different from the new ranking score computed on the server based on the search log analysis. PocketSearch resolves these conflicts by always adopting the maximum ranking score.

After the hash table has been updated, the server creates the necessary patch files for the database files that live on the phone. The new hash table and the 32 patch files are transmitted to the phone and the new cache becomes available to the user. Note that the amount of data exchanged between the phone and the server will usually be less than 1.5MB given that PocketSearch requires, on average, approximately 200KB for storing the hash table and 1MB for storing the search results (Figure 8).

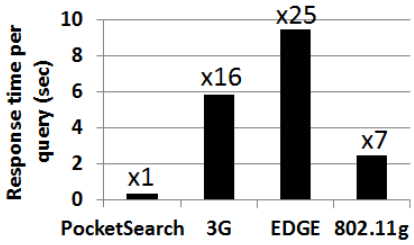
6. PocketSearch Evaluation

First, we use the prototype implementation to quantify the amount of time required to serve a search query through PocketSearch and compare its performance to that of the different radio links available on the phone. Second, we extract anonymized search query streams from the `m.bing.com` search logs and run them against PocketSearch to quantify what fraction of the query volume of an actual user can be served locally on the phone.

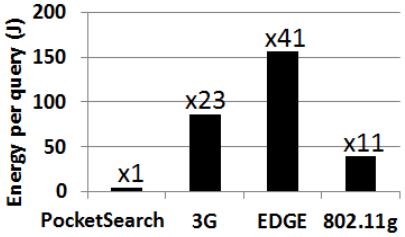
6.1 Cache Hit Performance

All of the measurements presented in this section were acquired using the prototype PocketSearch implementation on a Sony Ericsson Experia X1a cell phone running Windows Mobile 6.1 connected to AT&T's network. (Figure 1). To measure how fast queries are served using PocketSearch and the different radios available on the phone (Edge, 3G, 802.11g), we randomly selected 100 different queries for which cached search results were available. In every experiment, each of the 100 queries was submitted 100 times and the average user response time was calculated.

[3] The parameter λ controls how fast the ranking score is decayed.



(a)



(b)

Figure 15. (a) Average search user response time per query. (b) Average energy per query.

Operation	Average Time (ms)	Percentage
Hash Table Lookup	0.01	$\approx 0\%$
Fetch Search Results	10	2.7%
Browser Rendering	361	96.7%
Miscellaneous	7	1.7%
Total	378	100%

Table 4. PocketSearch's user response time breakdown

User response time is defined as the elapsed time from the moment that the query is submitted (search button is clicked in Figure 1) to the moment that the embedded browser object in the application has completed rendering the search results web page.

In the experiments where the PocketSearch cache was used to serve queries, a cache containing all the *query-search result* pairs that account for 55% of the cumulative *query-search result* volume over a period of several months was used (Figures 7 and 8). This cache included approximately 2500 search results occupying 1MB of flash space as described in Section 5.2.2.

6.1.1 Search User Response Time

Figure 15(a) shows the average user response time per query when the PocketSearch cache or one of the radios on the phone is used. On average, PocketSearch is able to serve a query 16 times faster than 3G, 25 times faster than Edge and 7 times faster than 802.11g. Note that even though 802.11g can provide a low user response time that is slightly higher than 2 seconds, it has a major drawback. Due to its high power consumption, 802.11g is rarely turned on and connected to an access point on a continuous basis. As a result, in practice, 802.11g is not instantly available and requires extra steps that introduce delay and unnecessary user interaction.

Table 4 shows the breakdown of PocketSearch's user response time in the case of a cache hit. 96.7% of the time it takes PocketSearch to serve a query is spent at the browser while rendering the search results web page. The time it takes the cache to locate and retrieve search results is approximately 10ms and it accounts for only 3.3% of the overall user response time.

Furthermore, the time it takes PocketSearch to look up its hash table and determine if a query is a cache hit or a cache miss is only $10\mu s$ (Table 4). Therefore, in the case of a cache miss, the overall user response time will only be increased by $10\mu s$, a negligible increase given that any radio on the phone requires several seconds to serve a search query.

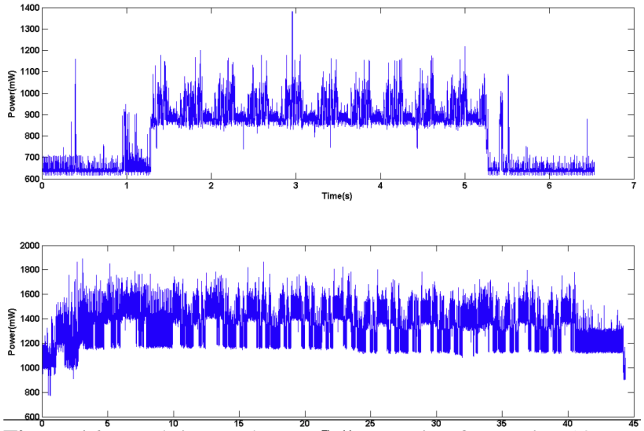


Figure 16. Total time and power consumption for serving 10 consecutive queries through PocketSearch (top) and the 3G radio (bottom).

	Navigation user response time		
	PocketSearch	3G	Speedup over 3G
Lightweight Page	15.378s	21.048s	28.7%
Heavyweight Page	30.378s	36.048s	16.7%

Table 5. Navigation user response time for PocketSearch and 3G for two example webpage load times.

6.1.2 Navigation User Response Time

By reducing search user response time, PocketSearch manages to also improve the overall navigation user response time, that includes both the time it takes the user to first search the web as well as the time to download the actual webpage. Table 5 shows the actual navigation time for two representative pages, a lightweight version and a heavyweight version that take 15 seconds and 30 seconds respectively to be downloaded and rendered over 3G. When PocketSearch serves the search results, the user can access the desired webpage up to approximately 29% faster than when querying through the 3G link.

6.1.3 Energy Consumption

Figure 15(b) shows the average energy consumed by the phone per query when PocketSearch or one of the radios on the phone is used. PocketSearch is on average 23 times more energy efficient than 3G, 41 times more energy efficient than Edge and 11 times more energy efficient that 802.11g. Note that the gap in the energy efficiency between PocketSearch and the different radios on the phone is larger than the corresponding gap in the user response time shown in Figure 15(a). This performance gap is justified by PocketSearch's ability to conserve energy in two ways (Figure 16). First, no data are being transmitted or received in the case of a cache hit, and thus the overall power consumption of the phone remains low (900mW vs 1500mW in Figure 16). Second, since PocketSearch achieves a user response time that is an order of magnitude lower compared to when the radios on the phone are used (4 seconds vs 40 seconds in Figure 16), the per query energy dissipation is significantly lower for PocketSearch.

6.2 Cache Hit Rate Performance

To quantify the cache hit rate achieved by PocketSearch for a typical user, we used anonymized search query streams from the mo-

User Class	Monthly Query Volume	% of Users
Low Volume	[20,40)	55%
Medium Volume	[40,140)	36%
High Volume	[140,460)	8%
Extreme Volume	[460,∞)	1%

Table 6. Classes of users and their characteristics.

bile search logs. To ensure a representative and unbiased selection of search query streams, we classified users in 4 different classes based on their monthly query volume. Table 6 shows the different user classes and the percentage of users in the mobile search logs that belongs to each class. Note that we ignore users that submit fewer than 20 queries per month for two reasons. First, PocketSearch is targeting users that frequently access the internet and search the web. Second, as higher-end smartphones with advanced browsing capabilities become more and more available, the average monthly query volume submitted by individual users will increase beyond the threshold of 20 queries per month.

For the experiments described in this section, we randomly selected 100 anonymized users from each class shown in Table 6 and extracted their search query streams from the mobile search logs over a period of one month. Each of the 400 search query streams was replayed against the PocketSearch cache that was generated using the mobile search logs of the preceding month. The resulting cache contained approximately 2500 search results that corresponded to 55% of the cumulative *query-search result* volume in the search logs. Note that the data used to build the cache and the data used to extract the 400 query streams were non-overlapping.

6.2.1 Hit Rate Results

Figure 17 shows the average hit rate for each user class described in Table 6. On average, 65% of the queries that an individual user submits are cache hits, and can be served 16 times faster. By examining Figure 17, it becomes apparent that the cache hit rate increases with the monthly query volume. PocketSearch achieves a cache hit rate of approximately 60% for the low volume class which immediately jumps to 70% for the medium volume class and to 75% for the high and extreme volume classes.

Figure 17 also shows the average cache hit rate for every user class in the cases where PocketSearch is using only either the community or personalization component of the cache. When only the community component of the cache is used, new queries and search results selected by the user are not cached over time and therefore, the cache cannot take advantage of the repeatability of mobile queries. When only the personalization component of the cache is used, the cache is initially empty and therefore cache hits are achieved only from repeated queries.

As Figure 17 illustrates, when only the community part of the cache is used, the average hit rate across all user classes is reduced from 65% to 55%. What is even more interesting is the fact that the hit rate seems to increase monotonically with the monthly query volume. Even though the exact same cache is used across all classes (since personalization is not used), the users that submit more queries seem to also experience higher hit rates.

When only the personalization part of the cache is used, the average hit rate across all user classes is reduced from 65% to 56.5%. Note that for every user class, the personalization part of the cache achieves the same or higher hit rate compared to the case where only the community part of the cache is used. This is another indication of the high repeatability of mobile queries that the personalization part of the cache is able to capture. In addition, the fact that the cache hit rate increases for users with higher query volumes due to the personalized component of the cache, shows

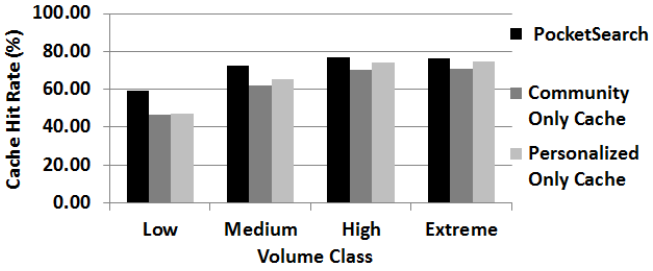


Figure 17. PocketSearch's average cache hit rate.

that users with higher query volumes repeat the same queries more often.

Even though users repeat mobile queries frequently, the community part of the cache is still very important for the overall user experience. Figure 18 shows the average hit rate for the different user classes during the first week (Figure 18(a)) and first two weeks (Figure 18(b)) of the one-month long query streams. Note, that after the first week, the hit rate of the personalization component of the cache remains lower than that of the community component of the cache. In particular, the fewer queries a user submits, the more time it takes the personalization component to warm up and be able to take advantage of the repeated queries. However, even during the first week, PocketSearch cache is able to provide the same hit rate with the one achieved in Figure 18 after a month. The community part of the cache provides a warm start for PocketSearch and the best possible out of the box search user experience.

The breakdown of the queries that result into a cache hit can be seen in Figure 19. On average and across all user classes, 59% of the cache hits are navigational queries[4] (*e.g.*, facebook, youtube, etc.) and the rest 41% are non-navigational queries that. The non-navigational hit rates are significantly increased or even doubled when compared to the medium volume class for both the high and extreme volume classes. This trend indicates that higher volume users tend to submit more diversified queries. However, even for this type of users PocketSearch is able to achieve high hit rates by taking advantage of the repeatability of mobile queries with its personalization component.

6.2.2 Daily Cache Updates

To understand how changes on the set of popular queries and search results on the server affect PocketSearch's cache hit rate, we repeated the same experiments while updating the PocketSearch cache on a daily basis as described in Section 5.4. On average, across all user classes PocketSearch achieves a cache hit rate of 66% when daily updates are used (we omit the graphs in the interest of space). This incremental improvement of 1.5% (66% vs 65% hit rate when daily updates are not used) is due to the fact that the popular set of queries and search results did not change significantly over the one month period we examined.

7. Architectural Considerations

The design and implementation of PocketSearch highlights the impact that the pocket cloudlet architecture can have on mobile user experience. Using only a couple of hundreds of KB of DRAM and a couple of MB of flash memory, PocketSearch can instantly serve 66% of the query volume that an average user submits. At the same time, Pocketsearch prevents 66% of the query volume across all users from hitting the cellular radio and the search engine servers, mitigating pressure on both cellular links and datacenters.

[4] these are queries that current browser cache substring matching techniques could also serve.

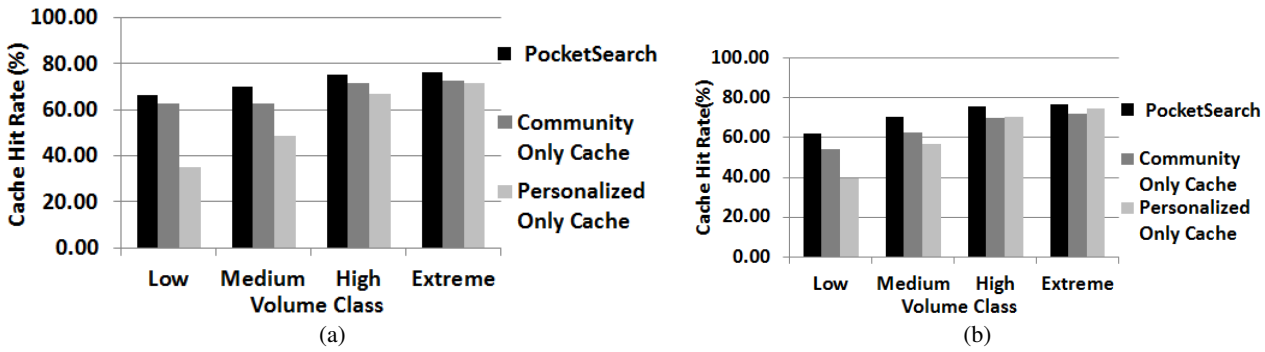


Figure 18. Average cache hit rate across the 4 user classes for (a) the first week and (b) the first two weeks of the month.

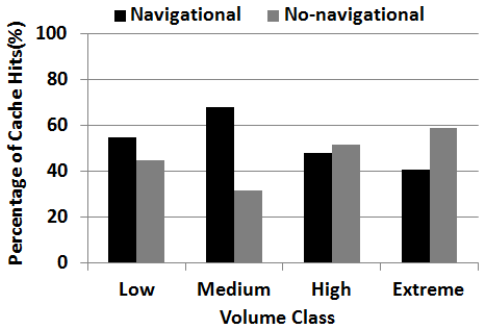


Figure 19. Breakdown of PocketSearch's cache hits into navigational and non-navigational across the 4 user classes.

Even though PocketSearch focuses on search services, we believe that the proposed architecture can serve as a template architecture for a broader family of cloudlets. Several other mobile cloud services beyond web search could leverage the same pocket cloudlet architecture, but each service eventually imposes its own memory requirements that might be very different when compared to PocketSearch. For instance, as the data in Table 2 shows, a mapping service cloudlet would require approximately 25 GB to cache all the map tiles for the user's state. Furthermore, creating a yellow pages cloudlet requires storing information about 23 million businesses across the United States, which according to Table 2 corresponds to approximately 100 GB. Similarly, web content, mobile ads and other pocket cloudlets impose their own requirements.

Even though each pocket cloudlet might share the same architecture and design principles, when multiple cloudlets with different requirements run on the same device, they naturally compete for resources within themselves and with other user applications. Managing the system resources properly across multiple cloudlets poses several architectural challenges.

User versus pocket cloudlets:The operating system will need to limit memory consumption such that enough memory is available to user data and applications. The more content cloudlets cache, the larger their indexes become. These indexes are stored in main memory and compete with regular user application memory usage. Pushing the indexes (or part of them) into slow storage-class memory would significantly affect cloudlet performance. We suggest the adoption of an intermediate tier consisting of fast storage-class memory, as described in Section 3, to address this problem.

Pocket cloudlet interactions: Many pocket cloudlet services cache related data. For example, when the user performs a web search, both search and ad cloudlets are invoked for the same query. In addition, the results will point to related web content, map tiles and yellow page entries. Different cloudlets have distinct storage requirements, and their relative storage allocation should take this into account. In addition, we believe that when memory needs to be reclaimed and cache entries evicted, it should be done in a coordinated fashion. If a particular query misses in the local search cache, there is not much benefit in hitting the ad cache because the latency bottleneck to service this query will be waking up the radio. Cache eviction policies should be managed by the operating system and coordinated such that closely related items are evicted together.

Security: Some cloudlets may include sensitive user and/or application data in their caches. Consequently, other cloudlets should not be allowed unrestricted access to those cache contents. For example, a map cloudlet shouldn't be allowed to access information regarding a user's recent bank transactions. We envision the operating system will provide such isolation and access control.

8. Related Work

The concept of caching information on client devices has been applied in the past in the context of web content [24], [21], [23], [25], [13] and advertisement [11] caching. The work presented in this paper differs in three ways. First, we describe a generalized mobile caching architecture that can be applied to various cloud services. Second, PocketSearch is complimentary to these efforts, in the sense that focuses on caching search results and not web content. Third, conversely to the previous work, PocketSearch was designed and implemented on an actual smartphone, addressing the challenges of building such an architecture on a mobile device.

File server caching systems, such as Coda [2], have focused on remote/offline access of files. Such schemes do not distinguish between a search result and a standard webpage HTML file let alone allow for personalized ranking of results within a page. Since these systems have no notion of search results, they don't take into account search trends into designing optimal client-side caching strategies. For instance, our data-driven approach indicated that by storing individual search results, storage requirements can be reduced by a factor of 8. In addition, PocketSearch combines both personal and community access characteristics to decide what search results to cache.

Approaches focusing on client side database caching have been combined with web browser caches to locally serve dynamic pages [1, 10]. Such approaches, however, aim to reproduce the exact same dynamic page as the servers (*e.g.*, search engines). They do not focus on assessing the popularity and cacheability of individual objects (*e.g.*, search results) within the dynamic page (*e.g.*, search results page) let alone personalize their ranking.

Content delivery or distribution networks (CDNs), such as the commercially available Akamai (`www.akamai.com`), aim to minimize user access time to content but they are complementary to PocketSearch. CDNs have great impact on wired networks where the latency between the client device and the nearby CDN node is low. However, on mobile devices, the bottleneck is the wireless link. By properly combining Pocket Cloudlets and CDNs, dis-

tributed cloud services that provide instant user experience can be created.

There have been several research efforts on understanding mobile search behavior through search log analysis [4], [3], [15], [16], [17], [18], [19], [30]. These efforts have analyzed search query volumes that vary from hundreds of thousands to several tenths of millions of queries. The search log analysis presented in this paper differs in two fundamental ways. First, we analyze 200 million mobile search queries, a query volume that is at least one order of magnitude larger than the query volume used in any other mobile search log study; volumes that large have been studied before only for desktop search [26]. Second, besides reporting similar observations on the locality of queries across mobile users, we also study in detail the repeatability of mobile queries for individual users.

The locality and repeatability of queries in the desktop search domain has been studied extensively and used to propose a server-side search caching scheme to reduce the load of search engines and improve user response time [20], [29], [7]. In the mobile domain though, the performance bottleneck is not the search engine but the slow radio link. Conversely to these approaches, PocketSearch is a client-based search cache that lives on the phone and enables search results to be displayed instantaneously as the user enters a query.

Previous work has also used mobile search log analysis findings to provide keyword auto-completion services on mobile devices [18] and thus facilitate and speed up query string entry. Such services are already popular on smartphone browsers and other mobile search applications and are complementary to PocketSearch, which focuses on providing fast search results and not query suggestions.

Recently, browsers and dedicated search applications on high-end smartphones such as Android and iPhone, have enabled web site suggestions as the user types a query. This is done in two ways.

First, for every new letter typed in the search box, a query is submitted in the background to the server for the partially entered query term and the most popular search result is returned as a website suggestion to the user. Note, that in this case a regular search query has to be submitted to the server over the radio link and therefore the usual slow mobile search experience is taking place.

Second, as the user types a query, a substring matching algorithm between the partial query string and all the website addresses in browser's cache can instantly provide the most relevant website that has been previously visited by the user. Unfortunately, this approach only works for a portion of the navigational queries.

9. Conclusions

In this paper, we proposed pocket cloudlets, an effective architecture that leverages abundant NVM in mobile devices to significantly improve user experience, both in terms of latency and battery life, by avoiding expensive radio wakeups and transmissions to access cloud services. We presented a case study of a search pocket cloudlet, and showed a surprising result: such service is viable even in today's mobile devices. We also explored other services that could benefit from a pocket cloudlet architecture and provided recommendations on how to build a system that supports multiple pocket cloudlets.

References

[1] E. Benson, A. Marcus, D. Karger, and S. Madden. Sync Kit: a persistent client-side database caching toolkit for data intensive websites. In *Proceedings of WWW*, pages 121–130, 2010.

[2] P. J. Braam. The CODA distributed file system. *Linux J.*, 1998, June 1998. ISSN 1075-3583.

[3] K. Church, B. Smyth, P. Cotter, and K. Bradley. Mobile information access: A study of emerging search behavior on the mobile internet. *ACM Trans. Web*, 1(1), 2007.

[4] K. Church, B. Smyth, K. Bradley, and P. Cotter. A large scale study of european mobile search behaviour. In *MobileHCI*, 2008.

[5] S. R. et al. Phase-change random access memory: A scalable technology. *IBM Journal of Reseach and Development*, 52(4/5), 2008.

[6] D. K. et all. A stackable cross point phase change memory. In *2009 International Electron Device Meeting*, Dec 2009.

[7] T. Fagni, R. Perego, F. Silvestri, S. Orlando, U. Ca, and F. Venezia. Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Trans. Inf. Syst*, 24, 2006.

[8] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, G. Ramesh, and D. Estrin. Diversity in smartphone usage. In *MobiSys*, 2010.

[9] I. T. R. for Semiconductors Working Group. International technology roadmap for semiconductors 2009 report. Technical report, International Technology Roadmap for Semiconductors, 2009.

[10] M. J. Franklin, M. J. Carey, and M. Livny. Local disk caching for client-server database systems. In *Proceedings of the 19th International Conference on Very Large Data Bases*, VLDB '93.

[11] S. Guha, A. Reznichenko, K. Tang, H. Haddadi, and P. Francis. Serving ads from localhost for performance, privacy, and profit. In *Hotnets*, 2009.

[12] Y. Huai. Spin-transfer torque mram (stt-mram): challenges and prospects. *AAPPS Bulletin*, 18(6):33–40, Dec 2008.

[13] S. Isaacman and M. Martonosi. The C-LINK system for collaborative web usage: A real-world deployment in rural nicaragua. In *NSDR '09*.

[14] R. C. Johnson. Memristors ready for prime time. http://www.eetimes.com/electronics-news/4077811/Memristors-ready-for-prime-time, Jul 2008.

[15] M. Kamvar and S. Baluja. A large scale study of wireless search behavior: Google mobile search. In *CHI*, 2006.

[16] M. Kamvar and S. Baluja. Deciphering trends in mobile search. *Computer*, 40(8):58–62, 2007.

[17] M. Kamvar and S. Baluja. The role of context in query input: using contextual signals to complete queries on mobile devices. In *MobileHCI*, 2007.

[18] M. Kamvar and S. Baluja. Query suggestions for mobile search: understanding usage patterns. In *CHI*, 2008.

[19] M. Kamvar, M. Kellar, R. Patel, and Y. Xu. Computers and iphones and mobile phones, oh my! In *WWW*, 2009.

[20] E. Markatos. On caching search engine query results. In *Computer Communications*, 2000.

[21] E. P. Markatos and C. E. Chronaki. A top-10 approach to prefetching on the web. In *Proceedings of INET*, 1998.

[22] Mobile Search Trends Report, `http://www.marketingcharts.com/interactive/mobile-local-search-ad-revenues-to-reach-13b-by-2013-8092/`.

[23] A. Nanopoulos, D. Katsaros, and Y. Manolopoulos. A data mining algorithm for generalized web prefetching. *IEEE Trans. on Knoweledge and Data Engineering*, 2003.

[24] V. N. Padmanabhan and J. C. Mogul. Using predictive prefetching to improve world wide web latency. *SIGCOMM Comput. Commun. Rev.*, 26(3), 1996.

[25] J. Pitkow and P. Pirolli. Mining longest repeating subsequences to predict world wide web surfing. In *USENIX*, pages 139–150, 1999.

[26] C. Silverstein, H. Marais, M. Henzinger, and M. Moricz. Analysis of a very large web search engine query log. *SIGIR Forum*, 33(1), 1999.

[27] J. Teevan, S. T. Dumais, and E. Horvitz. Personalizing search via automated analysis of interests and activities. In *SIGIR*, 2005.

[28] J. Teevan, E. Adar, R. Jones, and M. A. S. Potts. Information re-retrieval: repeat queries in yahoo's logs. In *SIGIR*, 2007.

[29] Y. Xie and D. O'Hallaron. Locality in search engine queries and its implications for caching. In *Infocom*, 2002.

[30] J. Yi, F. Maghoul, and J. Pedersen. Deciphering mobile search patterns: a study of yahoo! mobile search queries. In *WWW*, 2008.