HyRec: Leveraging Browsers for Scalable Recommenders

Antoine Boutet
INRIA
antoine.boutet@inria.fr

Davide Frey
INRIA
davide.frey@inria.fr

Rachid Guerraoui
EPFL
rachid.guerraoui@epfl.ch

Anne-Marie Kermarrec
INRIA
anne-marie.kermarrec@inria.fr

Rhicheek Patra
EPFL
rhicheek.patra@epfl.ch

ABSTRACT

The ever-growing amount of data available on the Internet calls for personalization. Yet, the most effective personalization schemes, such as those based on collaborative filtering (CF), are notoriously resource greedy. This paper presents *HyRec*, an online cost-effective scalable system for user-based CF personalization. *HyRec* offloads recommendation tasks onto the web browsers of users, while a server orchestrates the process and manages the relationships between user profiles.

HyRec has been fully implemented and extensively evaluated on several workloads from MovieLens and Digg. We convey the ability of *HyRec* to reduce the operation costs of content providers by nearly 50% and to provide a 100-fold improvement in scalability with respect to a centralized (or cloud-based recommender approach), while preserving the quality of personalization. We also show that *HyRec* is virtually transparent to users and induces only 3% of the bandwidth consumption of a P2P solution.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: Information filtering; H.3.5 [**Online Information Services**]: Web-based services; K.6.4 [**System Management**]: Centralization/decentralization

General Terms

DESIGN

Keywords

Personalization, Collaborative Filtering, Recommendation Systems

1. INTRODUCTION

Personalization has become an essential tool to navigate the wealth of information available on the Internet. Particularly popular now are recommendation systems which provide users with personalized content, based on their past behavior and on that of similar users. These systems have been successfully applied by major online retailers such as Amazon to propose new items to their customers. Social networks, such as Facebook, exploit them to suggest friends to users and to filter the content displayed on their feeds, while Google or Yahoo! use them to provide personalized news to users.

Yet, the need to personalize content is no longer an exclusive requirement of large companies. Personalization is arguably crucial for every web-content editor, including relatively small ones. A very typical example is that of VideoJeux.Com[1], a relatively small French online video-game magazine employing 20 people only, while gathering 3 million registered users (responsible for only 20% of the traffic: the rest being generated by unregistered ones who comment and discuss published content). The site of the company, visited 45 million times per month, enables access to 6,000 discussion forums, which in turn generate up to 300,000 messages per day. In many cases, the amount of information sometimes over-floods specialized users interested in specific games, who finally unregister. Clearly, the number of companies in similar situations is growing and all would greatly benefit from a personalization scheme providing users with recommendations about what they would most likely be interested in. However, state-of-the-art personalization solutions still represent a significant investment in terms of computing power and money.

The motivation of our work is to explore solutions that can "democratize personalization" by making it accessible to any content-provider company, without requiring huge investments. In this paper, we introduce *HyRec*, a *middleware hybrid* architecture capable of providing a cost-effective personalization platform to web-content editors. Instead of scaling either through larger and larger recommendation back-end servers, or through fully decentralized solutions that rely solely on clients, *HyRec* delegates expensive computation tasks to users' web browsers while, at the same time, retaining on the server side the system's management tasks and the maintenance of the graph reflecting the relationships between user profiles.

HyRec implements a *user-based collaborative-filtering* scheme (CF): it predicts the interests of a user by collecting preferences or taste information from many other users (collaborating) [30]. CF is content agnostic and represents a natural opportunity for decentralizing recommendation tasks on user machines. More specifically, *HyRec* adopts a k-nearest-neighbor (KNN) strategy, which consists in computing the k nearest neighbors according to a given similarity metric, and identifying the items to recommend from this set of neighbors [49]. The challenge is to cope with a large number of users and items. Traditional centralized recommendation architectures achieve this by computing neighborhood information offline and exploiting elastic cloud platforms to massively parallelize the recommendation jobs on a large number of nodes [25, 26]. Yet,

[1]We omit the real name for confidentiality reasons.

MIDDLEWARE '14, December 08 - 12 2014, Bordeaux, France

http://dx.doi.org/10.1145/2663165.2663315.

offline computation is less effective when new content is continuously added: forcing periodic re-computations induces significant costs [25, 38, 41].

HyRec's architecture avoids the need to process the entire sets of users and items by means of a sampling-based approach inspired by epidemic (gossip-based) computing [50, 19], and successfully used in state of the art KNN graph construction [28] as well as query processing [15].

The computation of the personalization operations of a user are performed transparently by the browser on the user's machine (which we sometimes simply call "the user" or "the client"). The *HyRec* server provides each user's browser with a sample set of profiles of other users (candidate set). Every browser then computes its user's KNN and most popular items based on this sample. The server uses, in turn, the user's new neighbors to compute the next sample. This iterative process implements a feedback mechanism that keeps improving the quality of the selected neighbors and leads them to converge very quickly to those that could have been computed using global knowledge. This is achieved without the need for *HyRec* to reveal the identity of any user to other users: the user/profile association is hidden through an anonymous mapping: periodically, the identifiers of the items and the users in the candidate sets are anonymously shuffled.

We fully implemented *HyRec* and its code is available [7]. We also extensively evaluated it in the context of two use cases: Digg, a personalized feed, and MovieLens, a movie recommender. We used real traces in both cases. We compared *HyRec* with solutions based on a centralized infrastructure as well as with a fully decentralized one. Our results show that the quality of the KNN approximation provided by *HyRec* is within 10% of the optimum for MovieLens. As the convergence of the KNN graph is driven by user activity, users who are frequently online benefit from a better approximation than users who are rarely online. We show that the reactiveness of *HyRec* in computing and refining the KNN graph during the activity of online users drastically improves recommendation quality with respect to solutions that use offline clustering, which may update this graph too late to provide useful recommendations.

HyRec reduces the server's computational cost by a factor ranging from 1.4 to 2. We show that, as the scale of the system increases, its ability to serve clients is much higher than that of a centralized approach (100-fold improvement). By computing KNN selection at the edge of the network on user machines, *HyRec* inherently reduces the operational costs incurred by the content provider. With our largest case study, *HyRec* yields a cost reduction for the content provider of nearly 50% with respect to a centralized solution. We show that the impact of *HyRec* on user machines is negligible, compared to a fully decentralized (P2P) solution. In the case of our Digg dataset for instance, a single user machine transmits around 24MB with the P2P approach, and only 8kB with *HyRec*. We also show that *HyRec* can exploit clients with small mobile devices without impacting user activities.

The rest of the paper is organized as follows. In Section 2 we introduce some background on collaborative filtering and recommender architectures. Section 3 and Section 4 present the design and implementation of *HyRec*. Our extensive evaluation of *HyRec* follows in Section 5. Section 6 concludes the paper.

2. BACKGROUND AND RELATED WORK

Before presenting *HyRec*, we provide some background on user-based collaborative filtering (CF) systems. We also recall the principles underlying centralized and decentralized CF recommendation systems (or simply recommenders) and motivate the very notion of a hybrid architecture.

2.1 Collaborative filtering

While content-based recommenders leverage content similarities between items, *user-based collaborative filtering* (CF in the following) focuses mainly on users [34]. CF recommenders build neighborhoods of users based on their interests in similar (e.g. overlapping) sets of items. Due to its content-agnostic nature, CF has now been adopted in a wide range of settings [30].

Notations. We consider a set of users $U = u_1, u_2, \ldots, u_N$ and a set of items $I = i_1, i_2, \ldots, i_M$. Each user $u \in U$ has a profile $P_u = < u, i, v >$, collecting her opinions on the items she has been exposed to, where u is a user, i an item, and v the score value representing the opinion of user u on i. For the sake of simplicity, we only consider binary ratings indicating whether a user *liked* or *disliked* an item after being exposed to it. This rating can be easily extended to the non-binary case [47].

Two steps are typically involved when a recommender provides a user with a set of items $R \subseteq I$: *neighbor selection* and *item recommendation*.

Algorithm 1 *KNN selection: $\gamma(P_u, S_u)$ where P_u is the profile of the user u and S_u is a candidate set for user u*

1: var $similarity[]$;
2: **for all** uid : user in S_u **do**
3: $\quad similarity[uid] = score(P_u, S_u[uid].getProfile())$;
4: **end for**
5: $N_u = subList(k, sort(similarity))$;
6: **return:** N_u, the k users with the highest similarity;

Neighbor selection. Neighbor selection consists in computing, for each user u, the (k) most similar users (referred to as KNN, k nearest neighbors, in the rest of the paper), with respect to a given similarity metric: we use cosine similarity [30] in this paper, but any other metric could be used. A brute force KNN computation has a complexity of $O(N^2)$ and designing low-complexity computations remains an open problem. We use a sampling-based approach to reduce drastically the dimension of the problem while achieving accurate results. The approach proceeds by successive approximations. At each iteration (Algorithm 1), it identifies a set of candidate users starting from the current KNN approximation, N_u. Then it computes the similarity between u's profile and that of each user in the candidate set and updates the KNN approximation by retaining the best candidates. This sampling-based approach is also a good candidate for decentralization as it does not require any global knowledge about the system. Indeed it is used in both centralized [28] and decentralized [19, 18] recommenders although the details of the computation of the candidate set may differ.

Algorithm 2 *Recommendation: $\alpha(S_u, P_u)$ where P_u is the profile of the user u and S_u is a candidate set for user u*

1: var $popularity[]$;
2: **for all** uid : user in S_u **do**
3: $\quad$ **for all** iid : item in $S_u[uid].getProfile()$ **do**
4: $\quad\quad$ **if** P_u does not contain iid **then**
5: $\quad\quad\quad popularity[iid] + +$;
6: $\quad\quad$ **end if**
7: $\quad$ **end for**
8: **end for**
9: $R_u = subList(r, sort(popularity))$;
10: **return:** R_u, the r most popular items;

Item recommendation. A user-based recommender makes recommendations based on the user profiles identified during the KNN-selection phase (Algorithm 2). In our context, we consider a sim-

ple scheme: The system recommends to a user u the r items that are most popular among the profiles in the candidate set identified when computing the KNN of u [25].

The presence of new users (*i.e* with empty profiles) and new items (*i.e.* without ratings) leads to the so-called *cold-start* issue. Solutions to this problem typically rely on application-specific techniques such as content-based recommendation and are therefore out of the scope of this paper. Similarly, for the sake of simplicity, we do not consider the retrieval of recommended items and we assume that items are hosted somewhere else.

2.2 Centralized architecture

A typical recommender follows a client-server interaction model (Figure 1), namely a centralized architecture. The server maintains two global data structures: A *Profile Table*, recording the profiles of all the users in the system and the *KNN Table* containing the k nearest neighbors of each user. Users interact with a web browser, which in turn communicates with the server to update the profiles, fill the KNN tables, and compute the recommendations.

In this setting, the server performs all computation tasks. These include running KNN selection and item recommendation for all users. Due to its high computational cost, the neighbor selection is typically performed periodically offline on back-end servers, while item-recommendation is achieved in real time. Clearly, the load on the server also depends on the number of concurrent (online) users. The web server, application logic, recommendation subsystem, and associated databases may leverage distribution, for example by delegating computation- and data-intensive recommendation tasks to a data center using cloud-based solutions like in Google News [25]. Yet, all active components remain under the responsibility of the website owner.

One of the main challenges underlying centralized CF architectures is scalability. This is particularly true for websites that enable users to generate content. Dimension reduction and algorithmic optimizations [33, 32], or sample-based approaches [28, 31, 27], partially tackle the problem. Yet they do not remove the need to increase computational resources with the number of users and items [45, 25, 22]. Even with massive (map-reduce) parallelization [26] on multicore [46, 40] or elastic cloud architectures [25], CF remains expensive in terms of both hardware and energy consumption [24, 41].

2.3 Decentralized architecture

A radical way to address scalability is through a significant departure from centralized (cloud-based) architectures, namely through fully distributed solutions [51, 48, 17, 42, 52]. A fully decentralized architecture, e.g. [19, 21, 18], builds an overlay network comprising all user machines, typically resembling a uniform random graph topology [35]. Users can join and leave the system at any time, e.g. due to machine failures or voluntary disconnections. No user has global knowledge of the system. Instead, each maintains her own profile, her local KNN, and profile tables. This allows a user to compute her own recommendations without further interaction with other users, or a server.

Decentralized solutions exploit a sampling protocol similar to the one in Algorithm 1. At each iteration, each user, u, exchanges information with one of the users, say v, in her current KNN approximation. Users u and v exchange their k nearest neighbors (along with the associated profiles) and each of them merges it with an additional random sample obtaining a candidate set. Each of them then computes her similarity with each user in her candidate set and selects the most similar ones (see Figure 1 and Algorithm 1) to update her KNN approximation. This process converges in a few cycles (e.g. 10 to 20 in a 100.000 node system [50]).

The absence of a server makes it possible to fully distribute KNN selection and item recommendation. However, fully decentralized solutions—like the one we implemented and compare with in Section 5.6—face important deployment challenges. They require users to install specific software that must manage their on/off-line patterns, while taking care of synchronization between multiple devices that may not be online at the same time. Moreover, they require substantial complexity to deal with NAT devices and firewalls. These limitations, combined with the inherent scalability of decentralized solutions, provide a strong motivation for a *hybrid* approach like ours.

2.4 Towards a hybrid architecture

A hybrid approach combines a centralized entity that coordinates tasks and manages the graph of relationships between users with processes that run on user machines to perform computationally intensive tasks. The central entity can effectively manage dynamic connections and disconnections of users, while retaining the main scalability advantages of decentralized solutions. Unlike [19, 21], *HyRec* allows clients to have offline users within their KNN, thus leveraging clients that are not concurrently online. Moreover, *HyRec* makes it possible to compute similarities with all the 2-hop neighbors at once, leading to faster convergence.

Hybrid approaches have already proved successful in various contexts. SETI@home [13] leveraged the machines of volunteers to analyze radio-telescope data whereas Weka [36] does something similar for data mining. A distributed Weka requires either a grid hosted by the content-provider, or an application server on the clients. In addition, Weka is oriented towards data analysis and does not provide a real-time personalization system. TiVo [16] proposed a hybrid recommendation architecture similar to ours but with several important differences. First, it considers an item-based CF system. Second, it does not completely decentralize the personalization process. TiVo only offloads the computation of item recommendation scores to clients. The computation of the correlations between items is achieved on the server side. Since the latter operation is extremely expensive, TiVo's server only computes new correlations every two weeks, while its clients identify new recommendations once a day. This makes TiVo unsuitable for dynamic websites dealing in real time with continuous streams of items. As we explain below, *HyRec* addresses this limitation by delegating the entire filtering process to clients. It is to our knowledge the first system capable of doing so on any user-based CF platform.

3. HYREC

As we pointed out, *HyRec* lies between cheap, but complex-to-implement, hard-to-maintain, decentralized solutions, and efficient, but potentially costly, centralized designs. It leverages the *locality* of the computation tasks involved in user-based CF schemes.

The bottom of Figure 1 shows *HyRec*'s hybrid architecture and its interaction pattern. The server delegates both KNN selection and item recommendation to user's web browsers using a sampling-based approach. Consider a user, u, that accesses a web page with recommendations. The server first updates u's profile in its global data structure. Then, it identifies a personalized candidate set for u, containing the profiles of candidate users for the next KNN iteration, and sends it to the browser, which executes a piece of JavaScript code attached to the web page. This code computes the recommended items, it performs the similarity computations between the local profile and the ones in the candidate set, and sends the results to the server. In the following, we detail this process by focusing on each component of the architecture.

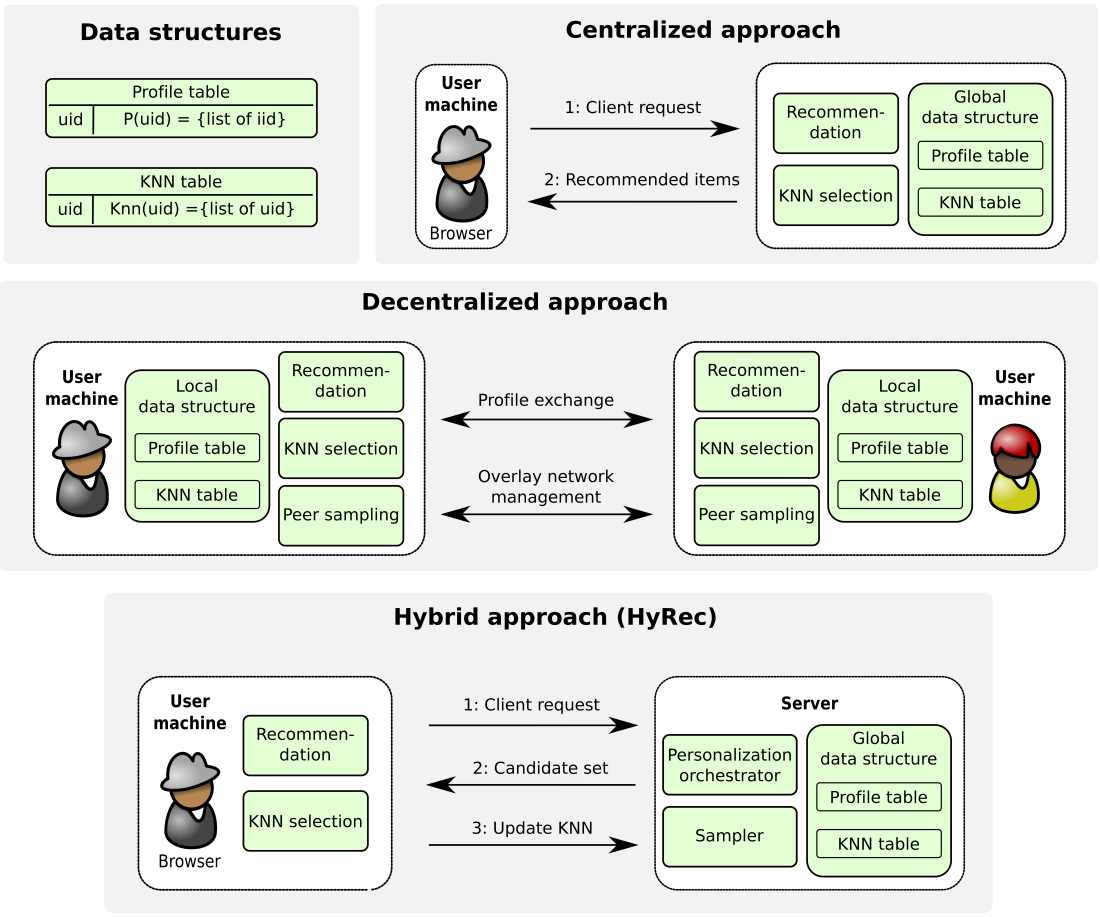

Figure 1: *Centralized, decentralized, and hybrid (*HyRec*) architectures for a recommender system.*

3.1 HyRec server

The server is in charge of *(i)* orchestrating the decentralization and *(ii)* maintaining the global data structures, a *profile table* and a *KNN table*. Each entry in the *Profile* and the *KNN* tables associates the identifier of each user respectively with her own profile and with those of the users in her current KNN approximation.

The server decomposes the recommendation process into *personalization jobs* that run on client-side widgets in the browsers of (connected) users. A *personalization job* essentially consists of a message containing the candidate set that allows the client to perform *(i)* KNN selection, and *(ii)* item recommendation by executing JavaScript code. In the following, we describe the two components of the server depicted in Figure 1: the *Sampler* and the *Personalization orchestrator*.

Sampler

HyRec relies on the sampling approach outlined in Algorithm 1 to associate each user with her k nearest neighbors. This *local* and *iterative* algorithm, inspired from epidemic clustering protocols [50, 19], leads to an online KNN-selection process, unlike the periodic one used in centralized architectures. This makes the system more reactive to dynamic interests.

The *sampler* participates in each iteration of the sampling algorithm and is responsible for preparing the *candidate set* (or *sample*). This consists of a small (with respect to the total number of users) set of candidate users, from which the client selects its next k nearest neighbors. Let N_u be a set containing the current approximation of the k nearest neighbors of u (k is a system parameter ranging from ten to a few tens of nodes). The sampler samples a candidate set $S_u(t)$ for a user u at time t by aggregating three sets: *(i)* the current approximation of u's KNN, N_u, *(ii)* the current KNN of the users in N_u, and *(iii)* k random users. Because these sets may contain duplicate entries (more and more as the KNN tables converge), the size of the sample is $\leq 2k + k^2$. As the neighborhood of u, N_u, converges towards the ideal one (N_u^*), the candidate set tends to get smaller and smaller as shown in Section 5.2.

By constraining the size of the candidate set, *HyRec*'s sampling-based approach not only limits computational cost, but also network traffic, while preserving recommendation quality as we show in our experiments. Research on epidemic [50] and k-nearest-neighbor graph construction [28] protocols show that the process converges very rapidly even in very large networks. Using u's neighbors and their neighbors provides the client with a set of candidates that are likely to have a high similarity with u. Adding random users to the sample prevents this search from getting stuck into a local optimum. More precisely, this guarantees that the process will eventually converge in the absence of profile changes by recording the user's k-nearest neighbors in the set N_u, so that $\lim_{t\to\infty} N_u - N_u^* = 0$, where N_u^* is the optimal set (*i.e.* containing the k most similar users). When profiles do change, which happens frequently in the targeted applications (e.g. news feed), the process provides each user with a close approximation of her current optimal neighbors.

Personalization orchestrator

The personalization orchestrator manages the interaction with the browsers. Once a user u accesses the server, (Arrow 1 in Figure 1), the orchestrator retrieves a candidate set, parametrized by k from the sampler and builds a personalization job. The personalization job for u consists of a message that includes u's profile and the profiles of all the candidates returned by the sampler (Arrow 2 in Figure 1). Finally, the orchestrator sends the personalization jobs, and updates the global data structures with the results of the KNN-selection iteration. Figure 2 illustrates the interactions between the clients and the server in *HyRec* as well as in a centralized approach.

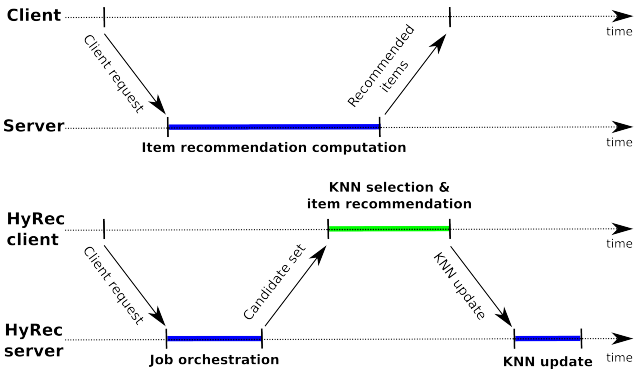

Figure 2: *Timeline: a centralized approach vs.* HyRec.

Clearly, sharing profiles among users may compromise their privacy. However, *HyRec* hides the user/profile association through an anonymous mapping that associates identifiers with users and items. *HyRec* periodically changes these identifiers to prevent curious users from determining which user corresponds to which profile in the received candidate set. As we discuss in Section 6, this mechanism does not suffice in the case of sensitive information (e.g., medical data) if cross-checking items is possible.

3.2 HyRec client

In *HyRec*, users interact with the recommender system through a web interface. The client side of *HyRec* consists of a javascript widget, running in a web browser. This widget serves as a web container that interacts with the server's web API. The *HyRec* widget sends requests to the server whenever the user, u, requires some recommendations. The server replies by providing a personalization job containing a candidate set. Upon receiving the job, the widget *(i)* computes u's recommended items and *(ii)* runs an iteration of the KNN-selection algorithm. Thanks to *HyRec*'s hybrid architecture, the widget does not need to maintain any local data structure: it receives the necessary information from the server and forgets it after displaying recommendations and sending the new KNN to the server.

Recommendation

Given the candidate set, S_u, and u's profile, P_u, the widget computes u's personalized recommendations as $R_u = \alpha(S_u, P_u)$, where $\alpha(S_u, P_u)$ returns the identifiers of the r most popular items among those that appear in the profiles in S_u, but not in P_u. These consist of the most popular items in S_u to which u has not yet been exposed.

As explained in Section 3.1, the candidate set contains the profiles of u's neighbors, u's two-hop neighbors, and k random users. By taking into account the items liked by the (one- and two-hop) neighbors, item recommendation exploits the opinions of similar users. By also taking into account items from the profile of random users, it also includes some popular items that may improve the serendipity of recommendations.

In a real application, once the item to be recommended have been identified, they might need to be retrieved from a web server to be displayed in a web page. We omit the process of retrieving the actual content of these items since this is application-dependent.

KNN selection

The client also updates the user's k-nearest neighbors. To achieve this, the KNN algorithm (Algorithm 1) computes the similarity between u's profile, P_u, and each of the profiles of the users in the candidate set, S_u. It then retains the users that exhibit the highest similarity values as u's new neighbors, $N_u = \gamma(P_u, S_u)$, where $\gamma(P_u, S_u)$ denotes the k users from S_u whose profiles are most similar to P_u according to a given similarity metric (here the cosine similarity). This data is sent back to the server to update the KNN table on the server (Arrow 3 in Figure 1).

4. HYREC IMPLEMENTATION

Our implementation of *HyRec* consists of a set of server-side modules and a client-side widget.

4.1 J2EE Servlets

Each component of the server consists of a J2EE servlet. These servlets come in two flavors: either as stand-alone components that can be run in different web servers, or bundled all together with a version of Jetty [9], a lightweight web server. Integrating the servlets into different customized web servers allows content providers to deploy our architecture on multiple hosts, thereby balancing the load associated with the various recommendation tasks (*e.g.* network load balancing). Bundling them with a Jetty instance makes it easy for content providers to deploy our solution into their existing web architectures.

4.2 Javascript Widget

The *HyRec* widget consists of a piece of JavaScript code attached to web pages, and can be executed without the need of special plugins. The widget's code use the jQuery implementation [10]. It identifies users through a cookie, collects user activities, runs KNN selection and item recommendation, manages the retrieval of the recommended items and displays them. All exchanges between the server and the widgets are formatted into JSON messages and compressed on the fly by the server using gzip (browsers are natively able to decompress messages). We use the Jackson implementation [8] to serialize JAVA objects to JSON message.

The use of JavaScript makes the operation of *HyRec* totally transparent to users thanks to the asynchronous nature of the AJAX model, which dissociates the display of a web page from the actions (i.e. javascript files, network communication) associated with it. Consider a web page that contains the *HyRec* widget: the browser will first display the web-page content, then it will asynchronously execute the personalization job and display the recommendation once all computations have completed.

4.3 Web API

The widget communicates with the *HyRec* server through the web API described in Table 1. The use of a public web API not only provides a simple way to implement our widget but also achieves authentication and makes it possible for content providers to build their own widgets that interact with *HyRec*. To develop a new widget, one simply needs to make the right API calls and tune

Web API	
`https://HyRec/online/?uid=uid`	Client request
`https://HyRec/neighbors/?uid=uid&id0=fid0&id1=fid1&...`	Update KNN selection

Customizable interfaces and methods	
`interface Sampler{...}`	Java interface on server side to define the sampling strategy
`setSimilarity();`	Customizable method on widget to determine the similarity metric for the neighbors selection
`setRecommendedItems();`	Customizable method on widget to determine the selected items to recommend

Table 1: Web API and main tools to customize *HyRec*.

the Javascript file associated with the *HyRec* widget. The content provider can tune the parts of the widget that process personalization jobs and collect user activities. For example, our implementation uses the cosine similarity metric and the most popular item-recommendation algorithm, but content providers can define new similarity metrics as well specify a new algorithm for item recommendation.

The current version of *HyRec* also integrates interfaces to customize parts of its behavior on the server-side. For example, content providers can control the sampling process and the size of user profiles, depending on the application, by means of the *Sampler* interface in Table 1. Similarly, they can include new fields in user profiles in the corresponding JSON messages returned by the API calls. All these settings are entirely transparent to end users.

5. EVALUATION

In this section, we show that *HyRec* meets ours goals: providing good-quality recommendations, reducing cost, and improving the scalability of the server as compared to a centralized approach; and this, without impacting the client, be it running on a smartphone or a laptop. We also show that *HyRec* uses less bandwidth than a decentralized recommender, besides not requiring custom software on user machines. We start with a description of the experimental setup. We then study KNN selection, recommendation quality, their impact on cost, and we evaluate the performance of *HyRec*'s server and client in comparison with alternatives.

5.1 Experimental setup

Platform

We consider a single server hosting all components (front and back-end) and assume that the database is entirely stored in memory. In practice, several machines can be used to implement each component separately to sustain the load at the network level. Yet, this does not affect the outcome of our experiments. We use a PowerEdges 2950 III, Bi Quad Core 2.5GHz, with 32 GB of memory and Gigabit Ethernet, to evaluate the server. To evaluate the client, we use both a Dell laptop latitude E4310, Bi Quad Core 2.67GHz with 4 GB of memory and Gigabit Ethernet under Linux Ubuntu, and a Wiko Cink King smartphone with Wi-Fi access running an Android system.

Datasets

We use real traces from a movie recommender based on the MovieLens (ML) workload [12] and from Digg [4], a social news web site. The ML dataset consists of movie-rating data collected through the ML recommender web site during a 7-month period and is often used to evaluate recommenders [25]. For the sake of simplicity, we project ML ratings into binary ratings as follows: for each item (movie) in a user profile, we set the rating to 1 (*liked*) if the initial rating of the user for that item is above the average rating of the user across all her items, and to 0 (*disliked*) otherwise. We use the three available versions of this dataset, varying in their number of users, to evaluate the quality of recommendation in *HyRec*.

Dataset	Users	Items	Ratings	Avg ratings
ML1	943	1,700	100,000	106
ML2	6,040	4,000	1,000,000	166
ML3	69,878	10,000	10,000,000	143
Digg	59,167	7,724	782,807	13

Table 2: *Dataset statistics.*

The Digg dataset allows us to consider an even more dynamic setting. Digg is a social news web site to discover and share content where the value of a piece of news is collectively determined. We collected traces from Digg for almost $60,000$ users and more than $7,500$ items over 2 weeks in 2010. This dataset contains all observed users in the specified period. Table 2 summarizes the workload. The average number of ratings per user for the Digg dataset is significantly smaller than for the ML datasets.

Competitors

We compare the performance of *HyRec* with that of several alternatives to highlight the benefits and limitations of our approach. The alternatives include both centralized and completely decentralized approaches. For centralized approaches, we distinguish two major families. *Offline solutions* perform KNN selection periodically on a back-end server, while they compute recommendations on demand on a front-end. *Online solutions* perform both KNN selection and item recommendation on demand on the front-end.

Metrics

View Similarity. To measure the effectiveness of *HyRec* in finding the nearest neighbors in term of interest, we compute the average profile similarity between a user and her neighbors, referred to as *view similarity* in the following. We obtain an upper bound on this view similarity by considering neighbors computed with global knowledge. We refer to this upper bound as the ideal KNN in the rest of the evaluation.

Recommendation Quality. To measure and compare recommendation quality, we adopt the same approach as in [37]. We split each dataset into a training and a test set according to time. The training set contains the first 80% of the ratings while the test set contains the remaining 20%. For each positive rating (*liked* item), r, in the 20%, the associated user requests a set of n recommendations, $\Re$. The *recommendation-quality metric* counts the number of positive ratings for which the $\Re$ set contains the corresponding item: the higher the better. If a positive rating represents a movie the user liked, this metric counts the number of recommendations that contain movies that the user is known to like.

System Metrics. To evaluate the performance of *HyRec*'s hybrid architecture, we measure execution time on both the *HyRec* server and the *HyRec* client. We also measure the bandwidth consumed

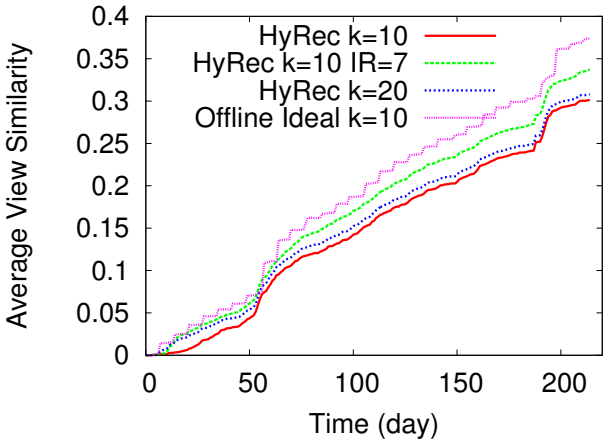

Figure 3: *Average view similarity on ML1 dataset for* HyRec *and ideal KNN.*

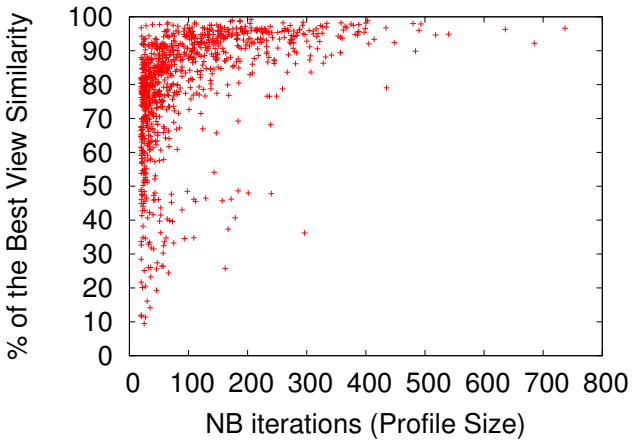

Figure 4: *Impact of the user's activity on the quality of the KNN selection of* HyRec *(ML1 dataset, $k = 10$).*

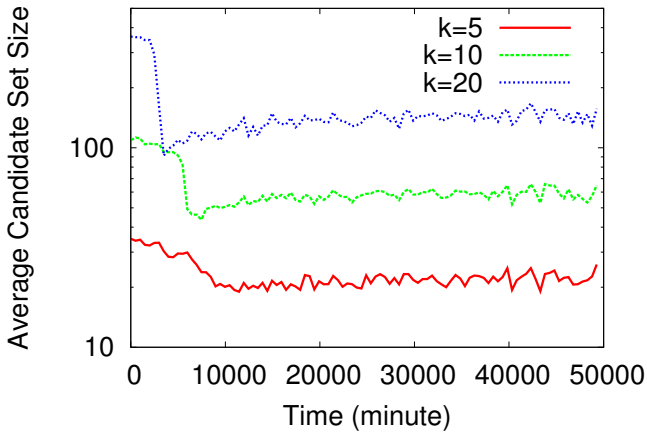

Figure 5: *Convergence of the candidate set size (ML1 dataset).*

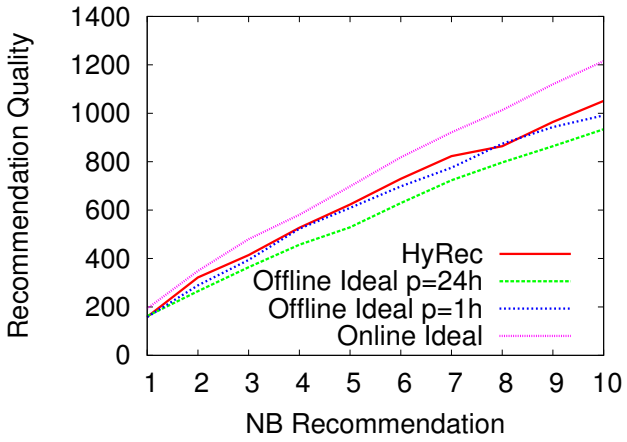

Figure 6: *Recommendation quality on the ML1 dataset for* HyRec *as well as offline and online ideal KNN ($k = 10$).*

during their exchanges. Finally, we measure the impact of *HyRec* on other applications running on a client machine as well as the impact of the load on the client machine on *HyRec*'s operation.

5.2 KNN selection quality

To evaluate the quality of the KNN selection provided by *HyRec*, we replay the rating activity of each user over time. When a user rates an item in the workload, the client sends a request to the server, triggering the computation of recommendations. We compare *HyRec* with the upper bound provided by the ideal KNN.

Figure 3 displays the average view similarity over all the users in the ML1 dataset as a function of time. Users start with empty profiles. As time elapses, they replay the ratings in the dataset, thereby increasing the average view similarity over time. The plot compares the results obtained by *HyRec* with those obtained by *Offline Ideal*. This consists of an offline protocol that computes the ideal KNN once a week. The period of one week allows us to identify a step-like behavior in the offline approach. This is because in offline protocols the neighbors remain fixed between two periodic computations and thus cannot follow the dynamics of user interests. A typical period in existing recommenders is in the order of 24h. Such a shorter period would make the steps thinner but it would not lead to faster convergence. Indeed, the upper bound on view similarity can be obtained by connecting the top-left corners of the steps in the offline-ideal curve on Figure 3. This upper bound corresponds to an online protocol that computes the ideal KNN for each recommendation. While interesting as a baseline, such a protocol is inapplicable due to its huge response times as we show in Section 5.5.

Overall, Figure 3 shows that *HyRec* effectively approximates this upper bound. For a neighborhood size of $k = 10$, *HyRec*'s average view similarity remains within 20% of that of the ideal KNN at the end of the experiment. The curve for $k = 20$ shows the impact of the neighborhood size: larger values of k result in larger candidate sets that converge faster to the nearest neighbors.

HyRec is an online protocol in the sense that it runs KNN selection as a reaction to user requests. The timing of such requests follows the information available in the data trace. As a term of comparison, we also consider a variant (IR=7) that bounds the inter-request time (*i.e.* the interval between two requests of the same client) to one week. Results show that the quality of KNN selection drastically improves according to the activity of users: more frequent user activity results in better view quality. An inter-request period of one week for $k = 10$ is enough to bring *HyRec*'s approximation within 10% of the upper bound at the end of the experiment.

To further analyze the impact of user activity on the KNN pro-

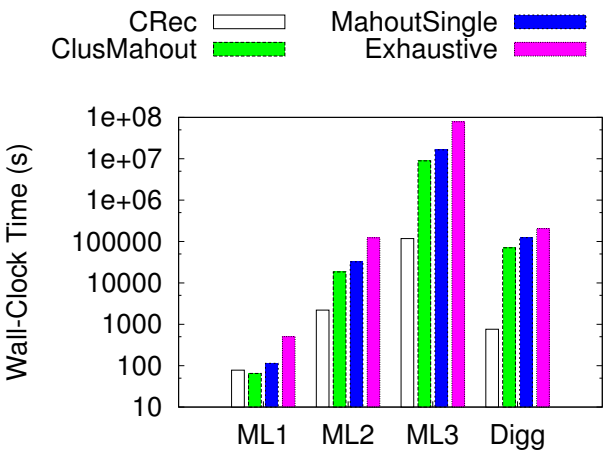

Figure 7: *Time to compute the k nearest neighbors on ML and Digg workloads.*

vided by *HyRec*, we plot the view similarity of each user as a percentage of her ideal view similarity (i.e. that of the ideal KNN) after replaying all the dataset. Figure 4 shows the results as a function of profile size. In our experimental setting, larger profiles imply more recommendation requests and thus more KNN iterations. Results clearly show the correlation between the number of iterations and the quality of KNN selection. More active users benefit from better KNN selection. The plot also shows that the vast majority of users have view-similarity ratios above 70%.

The iterative approach of *HyRec* refines its KNN selection over time. As the KNN of each user converge, the average size of the candidate set tends to decrease as each candidate is more likely to be a neighbor. Figure 5 depicts the average candidate-set size on the entire ML1 workload as a function of time for different values of k. We observe that the candidate-set size quickly converges to a stable value. For instance, for $k = 10$, its value quickly converges to around 55 instead of the upper bound of 120. The small fluctuations in the curve result from the continuous arrival of new users, who start with large candidate sets.

5.3 Recommendation quality

The recommendation process leverages KNN selection to identify the items to recommend as explained in Section 2. To evaluate this, Figure 6 plots the recommendation-quality metric against the number of recommendations provided in response to each client request. As described in Section 5.1, recommendation quality counts the number of recommendations that refer to positive ratings in the testing set [37]. For a fixed number of recommendations (x coordinate in the plot), higher recommendation quality implies both higher precision and higher recall [39].

Figure 6 compares *HyRec* with systems based on ideal KNN (both offline and online). The recommendation quality of offline approaches drastically changes according to the period of offline KNN selection (parameter p on Figure 6). The online-ideal solution, on the other hand, provides an upper bound on recommendation performance by computing the ideal KNN before providing each recommendation.

HyRec yields an up-to-12% improvement in recommendation quality with respect to the offline-ideal approach even when this runs with a period of 24 hours, which is already more costly than *HyRec* as we show in Section 5.4. It also provides better performance than offline ideal with a period of 1 hour and scores only 13% below the upper bound provided by online ideal.

Dataset	48h	24h	12h
ML1	8.6%	15.8%	27.4 %
ML2	31%	47.6 %	49.2 %
ML3	49.2 %	49.2 %	49.2 %
	12h	6h	2h
Digg	2.5%	5.0%	9.5%

Table 3: *Cost reduction provided by HyRec with respect to a centralized back-end server with varying KNN selection periods.*

To understand *HyRec*'s improvement on offline approaches, consider a user whose rating activity fits inside two cycles of offline KNN selection. This user will not benefit from any personalization with an offline approach. This is especially the case for new users, which start with random KNN. In *HyRec*, on the other hand, users start to form their KNN selection at their first rating and refine it during all their activity. This allows *HyRec* to achieve personalization quickly, efficiently, and dynamically.

5.4 Economic advantage of Hyrec

We now compare the cost of running the *HyRec* front-end with that of running several offline solutions based on the centralized recommender architecture depicted in Figure 1. In such solutions, a front-end server computes the recommended items in real time upon a client request, while a back-end server periodically runs the KNN selection. Since *HyRec* leverages user machines to run the KNN-selection task, it significantly reduces the cost of running a recommender system.

To ensure a fair comparison, we first identify a baseline by selecting the least expensive offline solution among several alternatives running on a cluster. *Offline-ideal* is the offline approach we considered in Sections 5.2 and 5.3. It computes similarities between all pairs of users thereby yielding the ideal KNN at each iteration. Offline-CRec is an offline solution that uses the same algorithm as *HyRec* (i.e. a sampling approach for KNN) but with a map-reduce-based architecture. Both exploit an implementation of the mapreduce paradigm on a single 4-core node [46]. Finally, *MahoutSingle* and *ClusMahout* are variants based on the user-based CF implementation in Mahout [11]. Widely employed to add recommendation features to application and websites, Mahout consists of a state-of-the-art open-source machine-learning library by Apache. Both *MahoutSingle* and *ClusMahout* exploit the Apache Hadoop platform [5] to parallelize KNN selection on multiple processing cores. *MahoutSingle* runs on a single 4-core node, while *ClusMahout* runs on two 4-core nodes. Because all four solutions share the same front-end, we only compare the running times of their KNN-selection tasks on the back-end. In all cases, we consider two periods for offline KNN selection: 48 hours on MovieLens and 12 hours on Digg.

Figure 7 depicts the results in terms of wall-clock time. Not surprisingly, we observe a strong correlation between the size of the dataset (in terms of number of users and size of the profile) and the time required to achieve KNN selection. We observe that *Offline-CRec* consistently outperforms other approaches on all datasets with the exception of ClusMahout using two nodes on the ML1 dataset. On average, *Offline-CRec* reduces the KNN-selection time by 95.5% and 66.4% with respect to Offline-ideal and ClusMahout, respectively. Moreover, the gap between the wall time required by *Offline-CRec* and by the other alternatives increases with the size of the dataset. We therefore select *Offline-CRec* as a baseline to evaluate the gains provided by *HyRec* in terms of cost.

Specifically we gauge the cost associated with running *Offline-CRec* and the *HyRec* front-end on a cloud infrastructure using Ama-

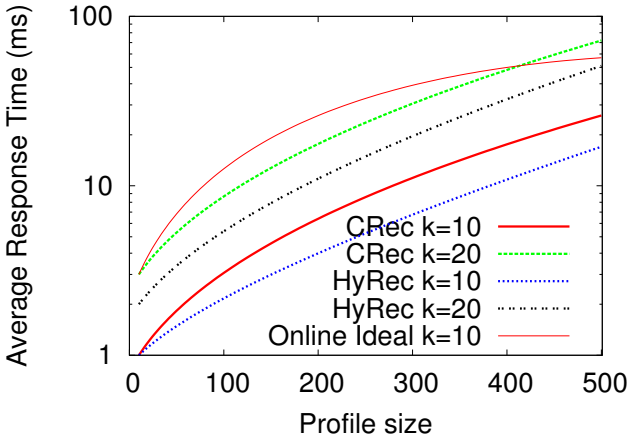

Figure 8: *Average response time for* HyRec *versus* CRec *according to the profile size.*

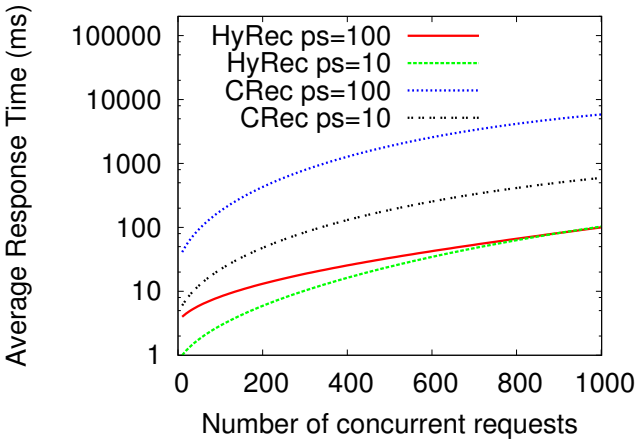

Figure 9: HyRec *vs* CRec *with a growing number of concurrent requests.*

zon EC2 services [3]. For the front-end server of both solutions, we consider the cheapest medium-utilization reserved instances which cost around $681 per year (the *Profile table* as well as the *KNN table* need to be stored in memory in order to answer client requests as fast as possible). For the back-end server of *Offline-CRec*, we consider one of the midrange compute-optimized on-demand instances with a price of $0.6 per hour (on-demand instances allow the content provider to be flexible in operating the offline KNN selection task). The efficiency of *Offline-CRec*'s KNN selection depends on the frequency at which it is triggered: a higher clustering frequency improves recommendation (as shown in Section 5.3) but it makes more frequent use of the on-demand instances, thereby increasing cost.

Based on these estimates, Table 3 considers back-end servers with varying KNN selection periods and summarizes the cost reduction achieved by *HyRec* over each of them. The numbers show the percentage of the total cost saved by the content provider. We do not consider extra costs for data transfer as the bandwidth overhead generated by *HyRec* is small and does not exceed the free quota even with the ML3 dataset.

The cost reduction provided by *HyRec* ranges from 8.6% for ML1 with a KNN selection period of 48 hours to 49.2% for ML3. To compute this last value of 49.2%, we considered a compute-optimized reserved instance over one year. This is cheaper than the number of required on-demand instances, and makes the cost of the offline back-end independent of the KNN selection period. Finally, we observe that the small cost reduction on Digg results from the small user profiles that characterize this dataset.

5.5 HyRec server evaluation

We now evaluate the performance of the *HyRec* server and its ability to scale when increasing the number of clients or the size of user profiles. In doing this, we consider *Offline-CRec* as a centralized alternative. However, in this section, we only consider its front-end server (referred to as *CRec*) and assume that its KNN table is up to date as a result of a previous offline computation.

The performance of *HyRec* mostly depends on the size of profiles and on that of the candidate set. Both these parameters are independent of the size of the dataset; thus we present our experiments on a single dataset, namely ML1, and we artificially control the size of profiles. The other datasets yield similar results. In addition, our experiments model the worst case by considering the largest possible candidate set for a given k (ignoring the decreasing size of the candidate set as the neighborhood converges). Finally, since KNN update messages from the client to the server are negligible when compared to the other messages, we ignore them in the evaluation.

Impact of the profile size.

The size of the user profile directly impacts the performance of the servers (*HyRec* and *CRec*). This is clearly application-dependent: for instance users tend to rate news articles more often than they rate movies. Typically, in *HyRec*, the larger the profile, the larger the size of the messages sent by the *HyRec* server to a *HyRec* client. In *CRec*, the profile size impacts the time spent to compute item recommendation: the larger the profile, the longer the item-recommendation process.

In order to evaluate the impact of the profile size, we run an experiment varying this parameter and evaluate the response time on the *HyRec* server and on the *CRec* front-end server. We use ab [2], a benchmark tool provided by Apache. Figure 8 plots the average (over 1000 requests) response time to serve a client request in *HyRec* and *CRec* with an increasing profile size. The response time for *HyRec* includes the time required to compress and decompress JSON messages. In spite of this, *HyRec* consistently achieves a better response time (by 33% on average) than *CRec* and this is clearer as the size of profiles increases. This can be explained by the fact that item recommendation on the *CRec* server takes consistently longer than *HyRec*'s personalization orchestrator takes to build messages.

Impact of the number of users.

The data in Figure 8 refers to a single request by a single user. However, the number of concurrent users definitely impacts the performance of *HyRec*. Figure 9 compares *HyRec* with *CRec* when facing a growing number of concurrent requests from users with profile sizes (ps) of 10 and 100. As expected, with smaller profile sizes, both *HyRec* and *CRec* serve requests more rapidly. Yet, *HyRec* consistently outperforms *CRec* regardless of the profile size. Results show that *HyRec* is able to serve as many concurrent requests with a profile size of 1000 as *CRec* with a profile size of 10. This represents a 100-fold improvement in the scalability of the front-end server for very large profiles.

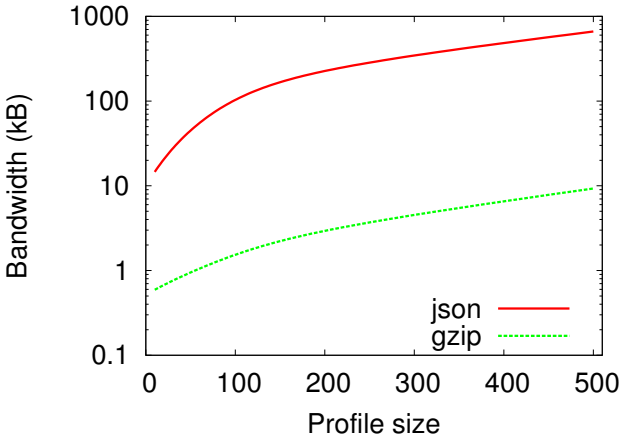

Figure 10: HyRec*: Profile size versus bandwidth overhead.*

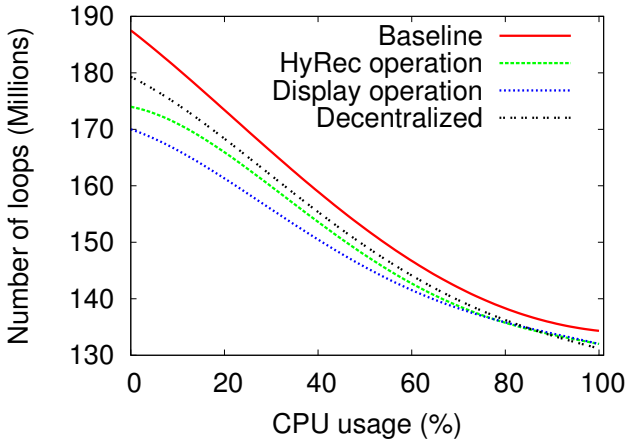

Figure 11: *Impact of* HyRec *widget, a decentralized recommender and a display operation on a client machine.*

Impact on bandwidth consumption.

Finally, the size of profiles impacts *HyRec*'s bandwidth consumption. Indeed, by delegating expensive computation tasks to clients, *HyRec* imposes some communication overhead with respect to a centralized architecture. Figure 10 shows the impact of profile size on the size of the JSON messages generated by the *HyRec* server upon a client request. Results show that the size of JSON messages grows almost linearly with the size of profiles. However, *HyRec*'s front-end server compresses messages on the fly through gzip. This results in a bandwidth consumption of less than 10KB even with a profile size of 500 (compression of around 71%). Note that bandwidth consumption also depends on the size of the candidate set. The size we consider here is an upper bound: the candidate set of a user quickly converges to smaller values. Overall, this shows that *HyRec*'s overhead is negligible when compared to the average size of a current web page (1.3MBytes [6]) and to the content of recommendations themselves, which can include pictures and text.

5.6 HyRec client evaluation

We now evaluate the cost of operating *HyRec* on the client. Our solution introduces a set of tasks on the client side, namely KNN computation, item recommendation, and sending update messages. No data structure needs to be maintained locally. This makes it possible for a user to use *HyRec* with the same profile from various devices. For *HyRec* to be sustainable, the operation of *HyRec* should not significantly impact the performance of a user's machine. Conversely, *HyRec* should be able to run on any device regardless of its load. We now show that *HyRec* complies with these requirements.

Impact of HyRec on a client machine.

We first measure the impact of operating the *HyRec* widget on an application running on a user's device (Figure 11). We consider a laptop on which we run a stress tool [14] to create a baseline level of CPU load (x axis in Figure 11). In parallel, we run a simple monitoring tool that executes an infinite loop consisting of a similarity computation. We measure the progress of this tool as the number of iterations it achieves over a given time window. This yields the *Baseline* curve in Figure 11.

For each of the other curves, we run an additional application in parallel with the stress and the monitoring tools. For *HyRec operation*, we run an infinite loop that continuously executes the operations of the *HyRec* widget: KNN selection and item recommendation with a profile size of 100. For *Display operation*, we run an infinite loop that requests some HTTP content (1,004 bytes from an RSS feed) from a server and displays it on a web page in a browser. Finally, for *Decentralized*, we run a fully decentralized recommender comprising P2P network management [35] as well as *HyRec*-like operations (KNN selection and recommendation).

Results demonstrate that the impact of *HyRec* on the client machine is minimal. *HyRec* affects the client machine in a similar way as requesting an item from an RSS feed and displaying it on a web page. The plot, also shows that the decentralized recommender has an even lower impact on the client. However, the impact of the decentralized recommender is stable over time since it is mostly due to overlay-network management. Conversely, the impact of *HyRec* is noticeable only while computing a recommendation. In addition, the operation of the *HyRec* widget is completely transparent to users, while a P2P recommender requires dedicated software and may encounter limitations related to churn and NAT traversal.

We also measured the impact of the *HyRec* widget, running in a browser, on other applications running on another tab of the browser while varying CPU usage. Results (not displayed here for space reason) show no impact of the *HyRec* computation job on other pages within the same browser. This is due to the fact that the browser considers each tab as a different process without links or shared resources. Overall, these experiments demonstrate the negligible impact of the *HyRec* widget on a user's browser.

Impact of CPU usage on the HyRec client.

We now evaluate to what extent the recommendation tasks of *HyRec* are impacted by the CPU load on the client machine on two different devices: a smartphone with Android using Wi-Fi and a laptop with Firefox using Ethernet. We measure the time spent by the widget within a browser with a profile size set to 100. To artificially impose load on client machines, we use the antutu smartphone benchmark [1] and stress [14] on the smartphone and the laptop, respectively. Figure 12 shows the average time required on client machines to execute the *HyRec* recommendation tasks depending on the baseline CPU usage on each of these two devices. We observe that even on a client machine with a CPU loaded at 50%, *HyRec* tasks run in less than 60ms on the smartphone and less than 10ms on the laptop. We also observe that this time increases only slowly on the laptop as the CPU gets more loaded. This conveys the fact that the *HyRec* widget can effectively operate even on highly loaded devices.

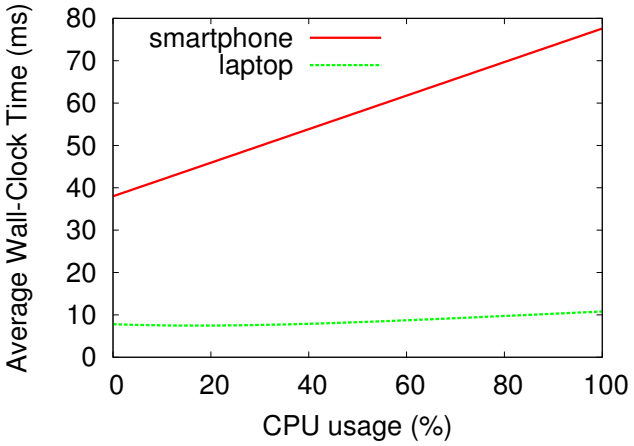

Figure 12: *Impact of the client machine load on the* HyRec *client.*

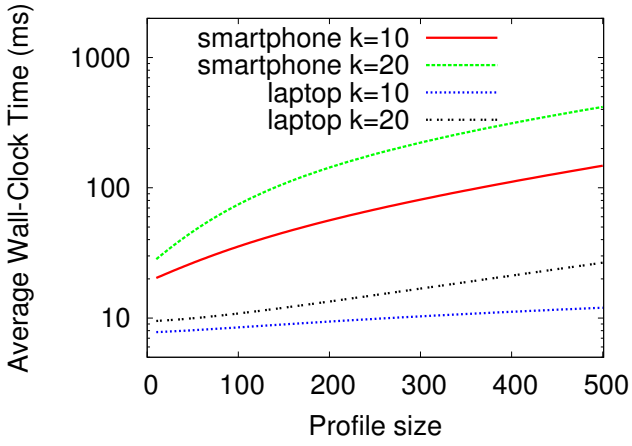

Figure 13: *Profile size's impact on the* HyRec *widget.*

Impact of profile size.

Unlike on the server, the impact of the size of user profiles on the *HyRec* widget is minimal. Figure 13 shows the duration of *HyRec* tasks (KNN selection and recommendation) on both a laptop and a smartphone with $k = 20$ and $k = 10$. Results show that the combined time for KNN selection and recommendation only increases by less than a factor of 1.5 and 7.2 for a laptop and a smartphone, respectively, with profile sizes ranging from 10 to 500 (Figure 13). We observe that although *HyRec* operations run faster on a laptop than on a smartphone, the impact of profile size remains limited, demonstrating that *HyRec* scales very well with large profiles.

Impact on bandwidth consumption.

In Section 5.5, we showed that the bandwidth consumption of *HyRec* is negligible even with large profiles. Here, we observe that, with respect to user machines, it is even lower than that of decentralized recommenders. While the operations carried out by *HyRec* are similar to those of a P2P recommender, the latter also needs to maintain an overlay network. This leads to continuous profile exchanges (typically every minute) that result in much higher traffic than *HyRec*, which only causes communication when responding to user requests. For example, on the Digg dataset (with an average of 13 ratings per user), each node in a P2P recommender exchanges approximately 24MB in the whole experiment, while a *HyRec* widget only exchanges 8kB in the same setting (3% of the of the bandwidth consumption of the P2P solution).

6. CONCLUDING REMARKS

We report in this paper on the design and evaluation of *HyRec*, the first user-based collaborative-filtering system with a hybrid architecture. The challenge for a hybrid system consists in deciding what to compute at the clients and what to maintain at the server in order to provide the scalability of P2P approaches while retaining a centralized orchestration. *HyRec* decomposes the KNN computation between server and clients in an original manner. Clients perform local KNN iterations, while the server stores intermediate results and makes them immediately available to all other clients. This not only provides scalability improvements with respect to centralized (possibly cloud-based) solutions, but it also overcomes the connectivity limitations of completely decentralized ones. We show that *HyRec* is cost-effective as it significantly reduces the cost of recommendation.

The motivation underlying *HyRec* is to democratize personalization by making it accessible to any content provider without requiring huge investments. *HyRec* is generic and can operate in many contexts. In its current version, it adopts a user-based CF scheme. However, it can use any data filtering algorithm that can be split among users' web browsers. Experimenting with several such algorithms in HyRec constitutes an interesting future perspective.

Applying *HyRec* in other contexts may also require additional optimizations. For instance, as shown in Section 5, the size of profiles can have an impact on the overall system performance. In our implementation, we compressed profiles through gzip (which provides an important compression ratio for a tag-based format such as JSON). But this might not always suffice. The size of user profiles depends on the very nature of the application as well as on the number of users in the system. The content provider may thus also constrain profiles by selecting only specific subsets of items, for example those rated within a specific time window.

Another important aspect is the Quality-of-Service perceived by the end user. A good Internet connection and a powerful device will provide recommendations much faster than a poor connection or an old device. However, even in sub-optimal conditions, the delay to display recommendations does not block the display of the rest of the web page thanks to the asynchronous nature of *HyRec*'s widget. Moreover, recent technologies like support for JavaScript threads in HTML5 [20] may further improve the performance of *HyRec* and encourage further exploration of hybrid web architectures.

Finally, the possibility of attacks and their potential impact can also be an important factor in determining whether to adopt a hybrid architecture. *HyRec* limits the impact of untrusted and malicious nodes: each user computes only its own recommendations. Furthermore, as we pointed out in Section 3.1, *HyRec* does not leak the browsing history or the exact profile-identity mapping of other users. De-anonymizing *HyRec*'s anonymous mapping is difficult if the data in profiles cannot be inferred from external sources such as social networks [44] or other datasets [43]. For privacy-sensitive applications such as recommending a doctor to a patient, it will however be interesting to enrich *HyRec* with stronger privacy mechanisms, such as homomorphic encryption [23], or to explore how to provide guarantees like differential privacy [29].

Acknowledgments. This work was partially funded by the French ANR project SocioPlug (ANR-13-INFR-0003), and by the DeSceNt project granted by the Labex CominLabs excellence laboratory (ANR-10-LABX-07-01).

7. REFERENCES

[1] Antutu. http://www.antutu.net.

[2] Apache http server benchmarking tool. http://httpd.apache.org/docs/2.2/programs/ab.html.

[3] Aws ec2 instances and pricing. http://aws.amazon.com/ec2.

[4] Digg. http://digg.com.

[5] Hadoop. http://hadoop.apache.org.

[6] Http archive. http://httparchive.org.

[7] Hyrec. http://gossple2.irisa.fr/~aboutet/hyrec.

[8] Jackson. http://jackson.codehaus.org.

[9] Jetty. http://www.eclipse.org/jetty.

[10] jquery. http://jquery.com.

[11] Mahout. http://mahout.apache.org.

[12] Movielens. http://www.grouplens.org/node/73.

[13] Seti@home. http://setiathome.berkeley.edu.

[14] Stress: a simple workload generator for linux. http://weather.ou.edu/~apw/projects/stress.

[15] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *EuroSys*, 2013.

[16] K. Ali and W. van Stam. Tivo: making show recommendations using a distributed collaborative filtering architecture. In *KDD*, 2004.

[17] L. Ardissono, A. Goy, G. Petrone, and M. Segnan. A multi-agent infrastructure for developing personalized web-based systems. *ACM TOIT*, 2005.

[18] R. Baraglia, P. Dazzi, M. Mordacchini, and L. Ricci. A peer-to-peer recommender system for self-emerging user communities based on gossip overlays. *Journal of Computer and System Sciences*, 79(2), 2013.

[19] M. Bertier, D. Frey, R. Guerraoui, A.M. Kermarrec, and V. Leroy. The gossple anonymous social network. In *Middleware*, 2010.

[20] J. Bipin. Multithreading in web pages using web workers. In *HTML5 Programming for ASP.NET Developers*. Apress, 2012.

[21] A. Boutet, D. Frey, R. Guerraoui, A. Jégou, and A.-M. Kermarrec. WhatsUp Decentralized Instant News Recommender. In *IPDPS*, 2013.

[22] M. Brand. Fast online svd revisions for lightweight recommender systems. In *SIAM ICDM*, 2003.

[23] J. Canny. Collaborative filtering with privacy via factor analysis. In *SIGIR*, 2002.

[24] Y. Chen, S. Alspaugh, D. Borthakur, and R. Katz. Energy efficiency for large-scale mapreduce workloads with significant interactive analysis. In *EuroSys*, 2012.

[25] A.S. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: scalable online collaborative filtering. In *WWW*, 2007.

[26] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 2008.

[27] E. Diaz-Aviles, L. Drumond, L. Schmidt-Thieme, and W. Nejdl. Real-time top-n recommendation in social streams. In *RecSys*, 2012.

[28] W. Dong, C. Moses, and K. Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *WWW*, 2011.

[29] C. Dwork. Differential privacy: a survey of results. In *TAMC*, 2008.

[30] M.D. Ekstrand, J.T. Riedl, and J.A. Konstan. *Collaborative Filtering Recommender Systems*. Now Publishers, 2011.

[31] A. Ene, S. Im, and B. Moseley. Fast clustering using mapreduce. In *KDD*, 2011.

[32] R. Gemulla, E. Nijkamp, P.J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *KDD*, 2011.

[33] R. Grover and M. J. Carey. Extending map-reduce for efficient predicate-based sampling. In *ICDE*, 2012.

[34] J.L. Herlocker, J.A. Konstan, A. Borchers, and J. Riedl. An algorithmic framework for performing collaborative filtering. In *SIGIR*, 1999.

[35] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based peer sampling. *ACM TCS*, 25(3), 2007.

[36] R. Khoussainov, X. Zuo, and N. Kushmerick. Grid-enabled weka: A toolkit for machine learning on the grid. In *ERCIM News*, 2004.

[37] J. J. Levandoski, M. Sarwat, A. Eldawy, and M. F. Mokbel. Lars: A location-aware recommender system. In *ICDE*, 2012.

[38] G. Linden, B. Smith, and J. York. Amazon.com recommendations: item-to-item collaborative filtering. *IEEE Internet Computing*, 2003.

[39] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, USA, 1999.

[40] Y. Mao, R. Morris, and M. F. Kaashoek. Optimizing mapreduce for multicore architectures. Technical Report MIT-CSAIL-TR-2010-020, MIT, 2010.

[41] D. Meisner, C.M. Sadler, L.A. Barroso, W.D. Weber, and T.F. Wenisch. Power management of online data-intensive services. *SIGARCH Comput. Archit. News*, 2011.

[42] B. N. Miller, J.A. Konstan, and J. Riedl. Pocketlens: toward a personal recommender system. *ACM TOIS*, 2004.

[43] A. Narayanan and V. Shmatikov. How to break anonymity of the netflix prize dataset. *CoRR*, 2007.

[44] A. Narayanan and V. Shmatikov. De-anonymizing social networks. In *SP*, 2009.

[45] Jia P. and Dinesh M. Bi-level locality sensitive hashing for k-nearest neighbor computation. In *ICDE*, 2012.

[46] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA*, 2007.

[47] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl. Grouplens: an open architecture for collaborative filtering of netnews. In *CSCW*, 1994.

[48] D. Rosaci, G.M.L. Sarné, and S. Garruzzo. Muaddib: A distributed recommender system supporting device adaptivity. *ACM TOIS*, 2009.

[49] X. Su and T.M. Khoshgoftaar. A survey of collaborative filtering techniques. *Adv. in Artif. Intell.*, 2009.

[50] S. Voulgaris and M. v. Steen. Epidemic-style management of semantic overlays for content-based searching. In *Euro-Par*, 2005.

[51] N. Zeilemaker, M. Capotă, A. Bakker, and J. Pouwelse. Tribler: P2p media search and sharing. In *MM*, 2011.

[52] S. Zhang, G. Wu, G. Chen, and L. Xu. On building and updating distributed lsi for p2p systems. In *ISPA*, 2005.