# ScaR: Towards a Real-Time Recommender Framework Following the Microservices Architecture

Emanuel Lacic
Graz University of Technology
Graz, Austria
elacic@know-center.at

Matthias Traub
Know-Center
Graz, Austria
mtraub@know-center.at

Dominik Kowald
Know-Center
Graz, Austria
dkowald@know-center.at

Elisabeth Lex
Graz University of Technology
Graz, Austria
elisabeth.lex@tugraz.at

## ABSTRACT

In this paper, we present our approach towards an effective scalable recommender framework termed *ScaR*. Our framework is based on the microservices architecture and exploits search technology to provide real-time recommendations. Since it is our aim to create a system that can be used in a broad range of scenarios, we designed it to be capable of handling various data streams and sources. We show its efficacy and scalability with an initial experiment on how the framework can be used in a large-scale setting.

## Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Information filtering*

## Keywords

real-time recommender framework; Apache Solr; scalability;

## 1. INTRODUCTION

Today's recommender systems need to satisfy multiple requirements such as high scalability and runtime performance in combination with state-of-the-art algorithms to produce accurate real-time recommendations. Additionally, recommender systems need to analyse huge amounts of data, process a high number of requests and handle dynamic streams of incoming data.

Up to now, already a number of open-source recommender libraries are available such as MyMediaLite[1], LensKit[2] or LibRec[3]. Although these libraries help developers and data scientists to implement, evaluate and adapt recommender approaches, it is mainly the industry giants such as Amazon and Netflix which apply recommender systems at scale.

[1]http://www.mymedialite.net/
[2]http://lenskit.org/
[3]http://www.librec.net/

*RecSys'15,* September 16–20, 2015, Vienna, Austria.
ACM xxx-x-xxxx-xxxx-x/xx/xx.
http://dx.doi.org/xx.xxxx/xxxxxxx.xxxxxxx.

**Objective.** Thus, the goal of our work is to provide real-time recommendations by *immediately* exploiting data streams within the requested recommendation approaches. To this end, we propose a flexible recommender framework termed *ScaR*[4], which adopts the *microservices architecture*[5] and which leverages the Apache Solr search engine. This framework enables us to:

(1) Generate real-time recommendations.
(3) Provide a scalable architecture to combine different recommender algorithms.
(4) Support large-scale offline and online evaluations.
(2) Support data streams and frequent updates without the necessity of time-expensive recalculations.

*ScaR* is freely available for the research community. At the same time it has already been used in a number of commercial projects, such as the TripRebel[6] hotel booking platform or the shopping portal ManouLenz[7].

The remainder of this paper is organized as follows: in Section 2, we discuss related work. In Section 3, we illustrate how our framework adopts the microservices architecture and outline the system with its core services and we show how the framework can be configured and deployed in a distributed manner. In Section 4, we describe how incoming data streams are handled. Finally, Section 5 shows the scaling capabilities of the framework with an experiment that incrementally increases the load of recommendation requests.

## 2. RELATED WORK

Most of the existing large-scale recommender systems, which focus on real-time recommendation (e.g., Netflix [2], Microsoft [12] and others [3, 17, 4]), use offline batch processing frameworks like Apache Hadoop, Apache Mahout, Spark [18] or GraphLab [10]. Other systems use relational database systems to provide real-time recommendations by querying recommendations from the generated data models [15]. However, in the literature, there is still a lack of work where data streams are handled and used *immediately* in the recommendation process. Moreover, query throughput and load balancing features are not clearly described or even ignored also in some framework documentations. For example, Apache Hadoop has only recently started to focus on interactive data processing via Apache Tez [13] which is an extensible framework to build high performance batch processing applications.

[4]http://scar.know-center.tugraz.at/scar/
[5]http://microservices.io/patterns/microservices.html
[6]http://www.triprebel.com
[7]http://www.manoulenz.tv/

Apart from that, most of the existing recommender systems focus on exploiting user behaviour, in form of user-item ratings, as their main information source for prediction. In respect to the prediction task, personalised recommendations using Matrix Factorisation approaches [7, 11] dominate the literature. However, all these model-based approaches need to be retrained whenever the data changes. Retraining tends to be very time-consuming when handling frequent and large-scale data updates. Empirical studies showed that especially for sparse data, a large number of factors is needed [14]. For instance, Diaz-Aviles et al. [5] proposed a method that improves Matrix Factorisation for real-time topic recommendation via a selective sampling strategy to update the model based on personalised small buffers. Compared to our work, they isolated the recommendation task for a specific data source, rather than including various contextual information, which has been shown to be very helpful when searching for relevant content [6]. Furthermore, in our work, we focus on integrating data updates in a large-scale scenario *immediately* as recommendations are requested, without the need to retrain the data model.

## 3. PROPOSED FRAMEWORK

We propose *ScaR*, an open-source, Java-based framework for generating real-time recommendation. The framework is also available via our Git repository[8]. The core concept of *ScaR* is its decomposition into several collaborating services, since it adopts the microservices architecture (recently also being adopted by Netflix). The architecture suggests to develop a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms. For this, at start-up, each service in *ScaR* spins up an embedded Jetty HTTP server, eliminating a number of unpleasant aspects of the deployment process (e.g., PermGen issues, application server configuration and maintenance, etc.[9]). The communication between the services is established through a standardised HTTP/REST interface, making each service easily exchangeable. The general architecture of *ScaR* is given in Figure 1, which shows the communication dependencies in a distributed environment between the five core services described below:

**Data-Modification-Layer (DML) Service.** The DML service handles all the communication (i.e., storing and querying data) with Apache Solr[10], the underlying data backend of the current version of the framework. Solr provides the capability of horizontal scaling by creating either shards (i.e., splitting the data into smaller indices to increase the performance of search queries for huge data sets) or replicas (i.e., cloning the existing shards to another machine to increase the fault-tolerance of the whole system). A beneficial feature of Apache Solr is its so called Near-Real-Time (NRT) mechanism. When activated, data will be processed and cached in memory and stored to the disk only after a certain period of time. For data that is rapidly changing or recently being added (e.g., user interactions), this feature dramatically increases the performance of the whole system. New data becomes available to the system the moment it gets uploaded, while the number of expensive disk write operations decreases.

Compared to many other recommender systems that only consider user-item information, Adomavicius et al. [1] introduced guidelines, which extend the traditional user-item paradigm with capturing of the context of recommendations by obtaining explicit, implicit or inferred information about the user. To support the proposed guidelines, *ScaR* is designed to be easily modified and adapted to handle various types of cpntexts, including item (e.g., ratings, description content, etc.), social (e.g., likes) and location data (e.g., geo-locations by means of latitude/longitude values). Moreover, having Solr as the data backend technology, *ScaR* can not only be used as a flexible recommender engine but also as a powerful search engine.

[8]https://git.know-center.tugraz.at/docs/?r=scar-framework.git
[9]http://www.dropwizard.io/getting-started.html
[10]http://lucene.apache.org/solr/

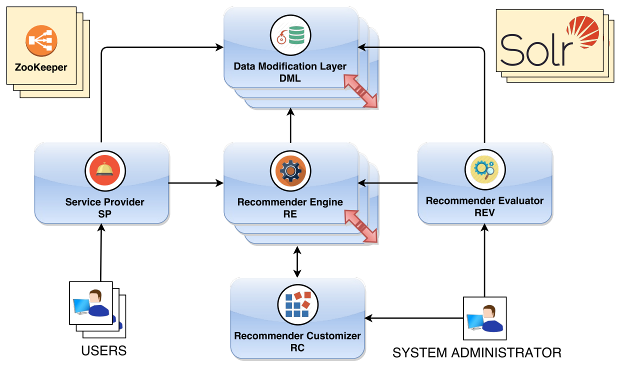

**Figure 1: The system architecture of ScaR for a distributed environment. Each service is a standalone HTTP server which knows the locations of its communicating partners with the help of ZooKeeper.**

**Recommender-Engine (RE) Service.** It is essential to provide fast recommendation to users, since most users are not willing to wait for a recommendation that was not explicitly requested in the first place. Since we use Apache Solr, we benefit from its internal data structure (i.e., inverted index) but also from its vector space representation for calculating similarities and to process large amounts of data in real-time. Besides, we utilize Solr's built-in *MoreLikeThis*[11] search functionality to retrieve similar items for our recommendations. Due to the direct communication between Apache Solr and the DML service, we can efficiently generate up-to-date recommendations that don't require computationally expensive pre-calculations (e.g., via batch jobs). *ScaR* is not only able to provide personalised recommendations, but also content-based and various hybrid combinations. The currently implemented RE service supports four types of algorithms: (1) user-based Collaborative Filtering (CF), (2) Content-Based Filtering (CBF), (3) Most Popular (MP), and (4) hybrid approaches that combine the former three (see also [9, 8] for more detailed implementation descriptions).

**Recommender-Customizer (RC) Service.** The RC service acts as a configuration and customisation repository for deployed recommender engines. Through so-called *recommender profiles*, we can parameterise our algorithms. For example, in case of CF, we could define in a profile the similarity measure (e.g., Jaccard or Cosine) or in case of hybrid approaches, the weightings of the algorithmic components. One can also go one step further and define the current context via post-filtering [1] of the recommended items (e.g., items from a specific location, time validity, etc.). This way, the same recommender algorithm can be customised for and be reused in different use cases. A key feature when updating the *recommender profiles* is the runtime synchronisation, meaning that all deployed RE services will be *immediately* notified by the RC service.

**Recommender-Evaluator (REV) Service.** The REV service provides the possibility to conduct large-scale offline and online evaluations of the implemented recommender algorithms. Offline evaluations (see also [9]) measure the algorithmic performance by

[11]https://cwiki.apache.org/confluence/display/solr/MoreLikeThis

a rich set of evaluation metrics (e.g., nDCG, Coverage, Diversity, Serendipity, Runtime etc.). To determine the real impact of the used recommender approaches under real-life conditions, online evaluations need to be conducted. Such evaluations have become increasingly popular in the industry, but are often not possible for researchers. To cope with that problem, especially in a large-scale setting, *ScaR* can be dynamically configured to perform a A/B or multivariate test during a predefined time interval. Besides manually starting any type of evaluation, just by deploying and using *ScaR*, the generated recommendations can be referenced when storing new user interactions, thus being able to analyse user feedback at any time.

**Service-Provider (SP) Service.** Although each service has its own REST API, direct access to these APIs should be restricted to admin users for security reasons. Hence, to ensure easy access to the framework, the SP service acts as a proxy for incoming, client-side HTTP requests. Furthermore, in a distributed environment, the SP also works as load balancer and re-routes incoming requests to the currently underused services. In case an A/B or multivariate test is initiated for the current time slot a recommendation is requested, the requested *recommendation profile* that parametrizes the recommender will be replaced by the *profile* assigned in the test set the requesting user was randomly assigned to.

**Service Orchestration.** To track and coordinate the deployed services, *ScaR* uses the Apache ZooKeeper Ensemble. By knowing the ZooKeeper locations, each service can scale horizontally, i.e., be deployed and started multiple times either on the same machine or on different ones. ZooKeeper coordinates the registered services by exposing a hierarchical namespace to deploy several different recommender domains on one system. Using the namespaces, services can be configured and deployed in the same server environment but still be isolated for different use cases (e.g., services dedicated for providing users with recommendations vs. services being used for evaluation).

## 4. DATA STREAM HANDLING

To handle data streams and frequent updates in real-time, *ScaR* makes use of Apache Solr's features, described below. We optimize the configuration of Apache Solr in respect to two key points:(i) frequency of incoming data (e.g., user interactions), and (ii) recommendation request rate. First, as new data arrives, the most expensive operation is writing the data into a stable storage, i.e., to the disk. Using Solr's *hard commit* functionality, these operations can be postponed to be automatically executed either after a defined time period or a maximal number of incoming data entries. Secondly, to be able to search the incoming data when generating recommendations in real-time, Solr's *soft commit* functionality needs to be configured to make the incoming data visible. As *soft commits* do not store the data to the stable storage, but rather make it visible, they are less-expensive. By adopting these *commit* functionalities it is possible for *ScaR* to support both, data intensive and recommendation intensive requests.

Still, commit intervals for soft and hard commit should be reasonable long to achieve the best performance. It usually takes some experimentation to find the right intervals, but from our previous experience, we set the *soft commit* interval between 1 to 5 seconds (i.e., data is visible to the recommender approach somewhere between 1 and 5 seconds after it arrived) and the *hard commit* interval on 1 to 2 minutes. If needed (i.e., the recommender system is experiencing more frequent and longer lasting data spikes), these intervals can be set to last even longer. To cope with the visibility problem of the incoming data when the *soft commit* interval is rather long, Apache Solr also provides a *real-time get* functionality.

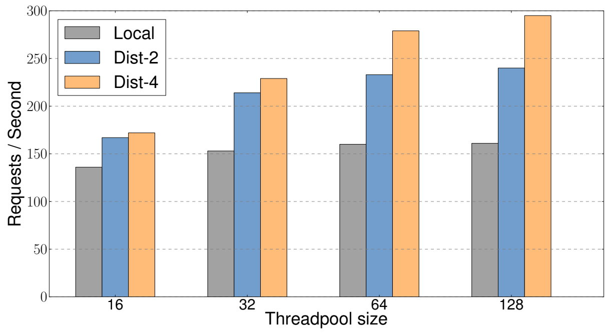

**Figure 2: Request processing rate of all 325,005 recommendation requests based on different request loads. *Dist-2* denotes a distributed setup with 2 *RE* and 2 *DML* services, as well as *Dist-4* a with 4 *RE* and 4 *DML* services. *Note:* each recommendation request calls all five implemented recommender algorithms.**

Using the *real-time get*, the latest version of any data entry can be *immediately* retrieved without the associated cost of the *commit*.

## 5. SCALABILITY EXPERIMENT

Traditionally, recommender systems deal with two types of entities, users and items. In respect to the scalability of *ScaR*, we used the Foursquare dataset provided by the authors of [16]. The dataset consists of $2,153,471$ users, $1,143,092$ venues, $1,021,970$ check-ins and $2,809,581$ ratings. We mainly focused on different Collaborative Filtering (CF) approaches, hybrid CF combinations and a Most Popular baseline. We have chosen user-based CF since it is not only a well-established recommender algorithm suitable for real-time recommendations, but also allows to incorporate various data features from different data sources, which has been shown to play an important role in making more accurate predictions [11].

To evaluate the performance of our framework, we performed three different deployment-based experiments under an exponentially growing workload. In each experiment, batch jobs were initiated to simulate a higher request rate (load) to the system, whereas the threadpool size was exponentially increased, having $16$, $32$, $64$ and $128$ threads simultaneously requesting recommendations. We focused on recommending items (i.e., venues in case of Foursquare), the most common type of recommendations. The initiated evaluation starts a continuous batch job, which fires recommendation requests for each user who rated at least $10$ items ($= 65,001$ users in total). For each user, five different recommendation approaches were requested: (1) a rating based Most Popular baseline; (2) a rating based CF; (3) a common check-in based CF; (4) a location network based CF (i.e., an artificially constructed network where ties between two users are existent if they visited the same location within the same day and hour); (5) a hybrid approach with arbitrary weights running and combining all four approaches in parallel. As a result, a total of $325,005$ independent recommendation requests were fired by the batch jobs.

The experiment was repeated three times, depending on the deployment method. First, a local deployment (*Local*) was made, i.e. every service was deployed once as a separate process on a server. In the second experiment (*Dist-2*), we scaled the framework horizontally by deploying an additional *DML* and *RE* service on another physical server. Last, to show that scaling the framework on the same machine (i.e., sharing the same resource) can also contribute to an increased performance, we deployed two more *RE* and two more *DML* services in addition to the previously deployed ones on the second server (*Dist-4*). To maintain the location information of each deployed service, we used two ZooKeeper instances.

## 5.1 Results

The mean processing runtime for each approach does not exceed 100 ms. Even the slowest processing time of the hybrid approach takes 199.8 ms, which is still feasible for recommendations required to be served in real-time. In respect to the processing power of the three deployment scenarios, Figure 2 shows the recommendation request processing rate in relation to the increasing request loads. The local deployment hits its limit at 161 recommendation requests per second for the last two heavy load simulations. This appears to be the upper bound and is due to the fact that the internal Jetty server needs some time to clean up already closed sessions and make them available to be reused again. Scaling the framework with another *RE* and a *DML* service on the second server increases the request processing rate by 49%. Adding two more services of the same kind results in processing 295 recommendation requests per second under a heavy load, being a 83% increase compared to the local deployment, showing that the framework's performance can be increased by horizontal scaling.

## 6. CONCLUSION

In this paper, we presented our approach and first experimental results towards a real-time recommender framework that adopts the microservices architecture. We showed how *ScaR*, in combination with the Apache Solr search engine, is suitable for a large-scale application setting where a high recommendation request load, together with frequent data updates needs to be supported without time-expensive recalculations. In order to evaluate the available recommendation approaches at large-scale, the *ScaR* framework also provides an evaluation service with offline as well as online evaluation methods.

In respect to future work, we plan to extend the framework with other backend solutions (e.g., a graph database) and provide separate services to be used by the recommendation approaches. This would allow us to conduct a comparative study between different backend technologies and their impact on recommender approaches. The result could be an increased runtime performance or even novel hybrid recommender approaches, since each approach would use the most suitable data backend. Moreover, we want to extend the framework and experiment on how to automatically tune the weights for the hybrid approaches using either different metrics (e.g., runtime, accuracy, diversity, novelty, etc.) or dynamically incorporating positive and negative recommendation feedback (e.g., recommendation click rate). Recently, Software-as-a-Service virtualization technologies (i.e., containers) have gained popularity as an alternative to using virtualization. Such lightweight resource containers provide several promising features, e.g. portability, more efficient scheduling and resource management, or less virtualization overhead. We plan a direct integration with one such technology, namely Docker, in our deployment scenario in order to investigate how it affects the frameworks' performance. Furthermore, we plan to conduct a large-scale user study to evaluate our approaches in respect to the cold-start problem during the next i-KNOW conference in October 2015.

**Acknowledgements.** This work is supported by the Know-Center and the EU-funded project Learning Layers (Grant Agreement : 318209). The Know-Center is funded within the Austrian COMET Program - Competence Centers for Excellent Technologies - under the auspices of the Austrian Ministry of Transport, Innovation and Technology, the Austrian Ministry of Economics and Labor and by the State of Styria. COMET is managed by the Austrian Research Promotion Agency (FFG).

## 7. REFERENCES

[1] G. Adomavicius and A. Tuzhilin. Context-aware recommender systems. In *In Proc of RecSys '08*.

[2] X. Amatriain. Big & personal: Data and models behind netflix recommendations. In *Proc. of BigMine '13*.

[3] S. Chan, T. Stone, K. P. Szeto, and K. H. Chan. Predictionio: a distributed machine learning server for practical software development. In *Proc. of CIKM '13*.

[4] C. Dai, F. Qian, W. Jiang, Z. Wang, and Z. Wu. A personalized recommendation system for netease dating site. *In Proc. of VLDB'14 Endow.*

[5] E. Diaz-Aviles, L. Drumond, L. Schmidt-Thieme, and W. Nejdl. Real-time top-n recommendation in social streams. In *Proc of RecSys '12*.

[6] G. Jones. Challenges and opportunities of context-aware information access. In *Ubiquitous Data Management '05*.

[7] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, Aug. 2009.

[8] E. Lacic, D. Kowald, L. Eberhard, C. Trattner, D. Parra, and L. Marinho. Utilizing online social network and location-based data to recommend products and categories in online marketplaces. In *MSM-MUSE '15*.

[9] E. Lacic, D. Kowald, D. Parra, M. Kahr, and C. Trattner. Towards a scalable social recommender engine for online marketplaces: The case of apache solr. In *Proc. of WWW '14*.

[10] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *In Proc. VLDB Endow. '12*.

[11] H. Ma, D. Zhou, C. Liu, M. R. Lyu, and I. King. Recommender systems with social regularization. In *Proc. of WSDM '11*.

[12] R. Ronen, N. Koenigstein, E. Ziklik, M. Sitruk, R. Yaari, and N. Haiby-Weiss. Sage: Recommender engine as a cloud service. In *Proc. of RecSys '13*.

[13] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *Proc. of SIGMOD '15*.

[14] R. Salakhutdinov and A. Mnih. Bayesian probabilistic matrix factorization using markov chain monte carlo. In *Proc. of ICML '08*.

[15] M. Sarwat, J. Avery, and M. F. Mokbel. Recdb in action: Recommendation made easy in relational databases. *Proc. VLDB Endow.*, 6(12):1242–1245, Aug. 2013.

[16] M. Sarwat, J. J. Levandoski, A. Eldawy, and M. F. Mokbel. Lars*: An efficient and scalable location-aware recommender system. *IEEE TKDE '14*.

[17] S. G. Walunj and K. Sadafale. An online recommendation system for e-commerce based on apache mahout framework. In *Proc. of SIGMIS-CPR '13*.

[18] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proc. of HotCloud'10*.