Cloud Based Real-Time Collaborative Filtering for Item-Item Recommendations

Rafael Pereira, Hélio Lopes, Karin Breitman
Departamento de Informática
PUC-Rio
Rio de Janeiro, Brazil
{rpereira, lopes, karin}@inf.puc-rio.br

Vicente Mundim, Wandenberg Peixoto
WebMedia
Globo.com
Rio de Janeiro, Brazil
{vicente.mundim, wandenberg}@corp.globo.com

***Abstract*—In this paper we argue that the combination of collaborative filtering techniques, particularly for item-item recommendations, with emergent cloud computing technology can drastically improve algorithm efficiency, particularly in situations where the number of items and users scales up to several million objects. We introduce a real-time item-item recommendation architecture, which rationalizes the use of resources by exploring on-demand computing. The proposed architecture provides a real-time solution for computing online item similarity, without having to resort to either model simplification or the use of input data sampling. We present results from a real life case study to show that it is possible to greatly reduce recommendation times (and overall financial costs) by using dynamic resource provisioning in the Cloud. Finally, we also discuss potential research opportunities that arise from this paradigm shift.**

Keywords - Distributed Architecture, Cloud Computing, System Architecture, Service Orientation, Collaborative Filtering

I. INTRODUCTION

The ability to offer relevant suggestions to users is becoming extremely relevant for web applications, particularly those whose business models are dependent on audience ratings, e.g., content commercialization and product sales [1]. Moreover, because effective recommendations play such an important role in user experience, users frequently resort to recommendations as a means for content discovery. This is the case with Amazon [2, 3] and Netflix [4, 5, 6].

Recommendation systems [7, 8] usually combine user profiles and item and product information to generate recommendations. In other words, a recommendation system is composed of contextual data, which is the information that the system has before it starts the recommendation process, input data, which is the information received for the recommendation process, and an algorithm that uses the context and input data to model recommendations.

Collaborative filtering [9, 10, 11, 12] is a recommendation technique that resorts to the user-item interaction history to find relationships between them. One example of such a relationship is computing the similarity between two items, such as videos [13], both viewed by the same group of users. In this case, no contextual information about the items is considered, which means that the recommendation algorithm does not have any information on "which are the items" and "what are their types or characteristics". There are several challenges associated with collaborative filtering engines. These algorithms must have sufficient performance to manipulate large sparse datasets, scale as the numbers of users and items grow, and retrieve relevant recommendations within a reasonable timeframe.

Thus, increasing the numbers of users and items poses great challenges for this type of system. One of these challenges is to improve the quality of recommendations made to users, since good recommendations increase user fidelity. A further challenge is the appropriate scaling of the system to retain its effectiveness. These two challenges tend to be conflicting; to obtain fast recommendations the model should be simpler, but this reduces the overall quality of the recommendations. Nevertheless, an efficient recommendation engine must take both aspects into consideration.

Additionally, there are specific scenarios that require the real-time processing of input data to produce relevant recommendations. This is the case in breaking news, where the content of interest is very volatile, and may become obsolete in a manner of minutes after being posted [14, 15]. In cases such as this, in additional to scalability, there is the challenge of producing relevant recommendations in real-time. This scenario adds an element of complexity to the recommendation models used by Amazon and Netflix, for example, which deal with less volatile items.

An ideal recommendation engine is capable of both scaling up, as the number of items and/or users grows, and producing a relevant recommendation in as small a time as possible (almost real-time).

One approach for scaling up the recommendation engine is through input data sampling. With this alternative it is possible to produce recommendations within a reasonable amount of time, using available, but often limited, computational resources. This process, of course, tends to reduce the accuracy of the recommendation, which may not be relevant to the user in question. Therefore, it is highly desirable that all available data is used to produce the best recommendation possible. The ability to contract computing resources on demand, from a cloud computing vendor, lifts the resource availability limitation, and thus enables the

development of a high quality solution, where all available input data is used to produce recommendations, while at the same time, allowing for the system to be scaled up (or down) to adapt to fluctuations in demand [16, 17, 18, 19].

It is important to note, however, that cloud computing environments provide the necessary resources to guarantee that all computation can be done. What is not guaranteed is that the computation can be done time effectively, i.e., producing real-time recommendations irrespective of the numbers of items and users in question. This is the challenge and the major contribution of this work.

Main Contributions. The goal of this paper is the proposal of a new general architecture for a real-time collaborative filtering system that produces item-item recommendations, since traditional approaches are not able to address scalability and real-time processing issues without any data or model simplification. The proposed architecture makes use of all available input data, as opposed to data sampling, to produce relevant recommendations. It also makes use of on-demand computing resources to scale up and down, and adapt to variations in the number of items as well as users, thereby meeting the needs of large content portals, i.e., those dealing with several million hits on a daily basis and trading with catalogs with millions of different items.

Paper Outline. In the next section, we introduce the key concepts to generate Item-Item recommendations through collaborative filtering, and the challenges associated with this process. Section III presents the proposed architecture and describes how the issues associated with such process are addressed. Section IV discusses how the proposed architecture was deployed in the Cloud, in a real production environment, and the results obtained with those tests. Finally, section V presents the conclusions, discussing further works.

II. ITEM-ITEM RECOMMENDATION

It is common in collaborative filtering systems to store user profiles as vectors of items and ratings. These vectors tend to increase continually as users interact with the system. Some systems take into account temporal dynamics to discount the standard deviations in user interests over time [20, 21]. User feedback may be binary, e.g., *liked* or *not liked*, or valued according to preference. Netflix [4, 5, 6], MovieLens [22], and Amazon [3] are among those that adopt the latter approach.

The two most common ways of implementing a solution for collaborative filtering are the use of neighborhood algorithms or graphs, and latent factorization models.

Neighborhood algorithms focus on extracting the relationship between users or items to build a model based on a graph capable of describing these adjacencies.

- Item-item methods build a neighborhood graph with vertexes connecting similar items.
- User-user methods build a neighborhood graph with vertexes connecting users with similar preferences.

The idea behind an item-item recommendation engine is, for any given item, finding a set of items that is most similar to the item in question. Similarity is measured using a combination of input data, generally structured in a bi-dimensional matrix with the first dimension representing users and the second items. The ultimate goals of an item-item recommendation system are to predict how users would rate an item that is not yet rated, and to recommend items from the collection.

With respect to user ratings, recommendation engines may use different types of feedback. Ideally, explicit feedback is preferred, with users explicitly indicating their preferences. However, in most situations, such information is not available. In these cases, it is necessary to infer user preferences through implicit feedback [23], which indirectly represents a user's preferences based on his/her behavior. Implicit feedback is obtained from user navigation history, the list of products bought, and even from the mouse path on specific screens.

After obtaining user feedback, for example, information that a user accessed a particular item, the input can be stored in a square matrix representing users and items, where, in this case, each element denotes whether an item was accessed by a user. Thus, considering each item separately, we have a multidimensional vector with each dimension representing a user. Consequently, as the number of distinct users in the system increases, the number of dimensions in the vector also increases. Typically, large e-commerce or content portals have tens or even hundreds of millions of unique visitors per month, which means that their item vectors will have a similar number of dimensions.

Therefore, the problem of obtaining the similarity between two specific items can be translated into one of obtaining the similarity between the two multidimensional vectors representing the items. One of the possible approaches for doing this is by calculating the cosine between these vectors. Having calculated the similarity between a specific item and all others, one can then obtain the items that are most similar to one another, and thus, create an item-item recommendation that takes into account the feedback from users. However, it is important to remember that for each piece of feedback received, for example, for each accessed item, the similarity between this and the other items can change, and hence the similarity between the current item and all the others in the vector must be recalculated.

In this scenario, considering an environment with millions of users and millions of items it would be necessary, for each piece of feedback received, to recalculate the cosine between two vectors with millions of dimensions to ensure the similarity graph is updated in real-time. This is necessary because with each feedback, one of the vectors' dimensions changes, which in turn changes the cosine value and the similarity between this vector and all other vectors in the vector space.

As an illustration consider a video website with N different videos (items), and M unique viewers (users). This information can be represented by an MxN matrix in which items are represented as columns and users as rows. This

matrix can be used to track feedback, i.e., users who have demonstrated an interest in the items. Whenever there is a new entry, the matrix must be updated and the similarity graph recalculated. Figure 1 depicts a concrete case where user U_3 plays video I_3. In this case element U_3xI_3 of the feedback matrix must be updated, and all item pairs I_3:I_1, I_3:I_2 … I_3:I_n must have their similarity recalculated as shown. For a model with 2 million items, this means 1,999,999 similarity calculations every time an item is viewed.

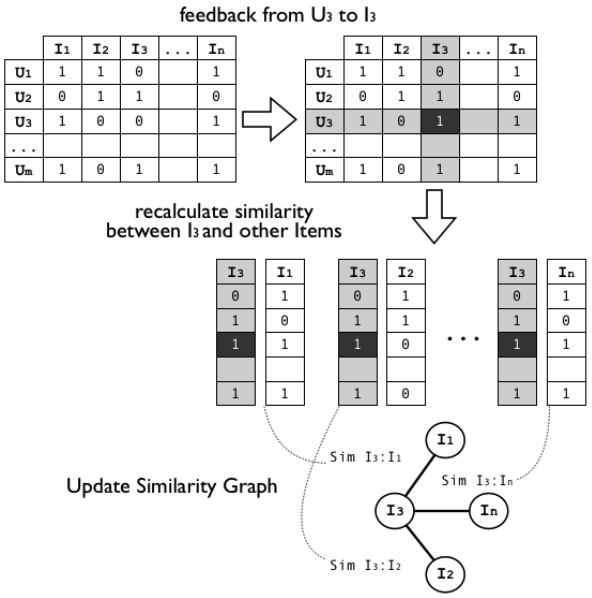

Figure 1. Similarity recalculation process for a single piece of feedback.

Another challenge associated with item-item recommendation is the variation in item relevance over time, that is, the introduction of temporal dynamics [24]. In the real world, the perception and popularity of an item is constantly changing as new items are introduced. Similarly, user preferences evolve. Thus, the system must take into account temporal effects in mapping the dynamic and variable nature of the interactions between users and items.

In practice, it is possible to associate a lifetime with each feedback, item, and user, so that the values of each dimension of the vector representing the item can be adjusted according to the temporal variation. Thus, a positive feedback could become neutral over time, given that this feedback does not necessarily represent current user behavior, or even given that this item is no longer relevant in the context.

Considering all aspects, a theoretical model to obtain item-item recommendations could be implemented by structuring the users and items in a two-dimensional array containing the feedback, and by calculating the cosine between vectors of items to obtain the similarity graph between items. Moreover, considering the temporal dynamics, which in the case of content providers, especially breaking news, is essential, we would need a third dimension in this matrix to define a life span for each piece of feedback or group thereof.

III. Platform Architecture

Based on the theoretical model for item-item similarity discussed in the previous section, we initially used a feedback matrix with N columns, where N is the number of items, and M rows, where M is the number of unique users who have given some kind of feedback about a particular item. We also discussed in the previous section that, for practical cases such as major content portals like CNN, BBC, and so on, there are millions of unique users and items, which would result in a matrix with trillions of elements. Thus, the first practical restriction in using such a model is the data representation. A practical representation requires an efficient model, which is feasible since this scenario would have a highly sparse matrix, as most users would not provide feedback, even implicit feedback, for many of the items.

The second challenge is associated with the real-time update of the similarity graph. In the theoretical model, for each piece of feedback it would be necessary to recalculate the cosine between the vectors representing the item associated with the feedback and all other items. If, in calculating the cosine, it were necessary to go through all the vector dimensions, one would have, at worst, an algorithm with $O(N^2M)$ time complexity for assembling the entire similarity graph [2]. This algorithm is given below. However, taking into account that the cosine between two vectors is represented by the dot product and magnitude, it is possible to obtain in practical terms, an algorithm with O(NM), since most users only provide feedback for a very limited number of items.

ALGORITHM I

For each existing item ***I1***
  For each user ***U*** *who performed a feedback for* ***I1***
    For each item ***I2*** *with a feedback from* ***U***
      Record that a user sent a feedback for ***I1*** *and* ***I2***
  For each item ***I2***
    Compute the similarity between ***I1*** *and* ***I2***

One of the simplest kinds of feedback that can be collected is user clicks, i.e., if a user takes action to access specific content (item). The mere fact that the user accesses an item indicates deliberately, even minimally, his interest in the item, which can be considered in a recommendation model. In this case, the engine would have binary feedback, representing whether the user accessed a particular item, for example, watched a video. A simpler and less expensive alternative for representing the same information contained in the feedback matrix in this situation could be through sets, where each item comprises a set of users who have accessed this item. In this way, the system would only keep the actual feedback, since 94% of the feedback matrix elements comprise unknown data [25], i.e., the absence of feedback.

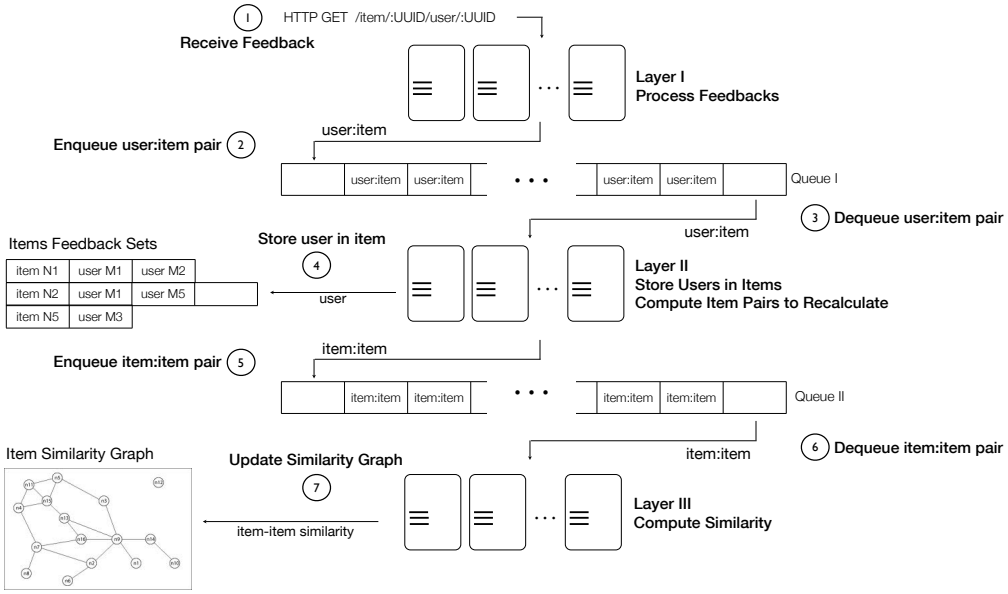

Figure 2. Multi-layer collaborative filtering process.

The representation of item vectors based on their feedback set is extremely useful in calculating the cosine similarity between items. With this type of binary feedback, the calculation of the scalar product necessary to obtain the cosine of the vectors can be accomplished through the intersection of the sets representing the items. Thus, its value would be equal to the number of existing elements in this intersection. Furthermore, the magnitude is the square root of the number of elements in the set. Thus, in practical terms, the use of sets to represent the feedback matrix reduces the number of operations, since the size of a set tends to be much smaller than the number of unique users in the model, which is the number of item vector dimensions.

Despite the simplification of the data used to represent the feedback, there is still the scalability challenge, i.e., how to obtain recommendations quickly and efficiently even if the numbers of users and items grow significantly. It is of particular interest to ascertain how to leverage the benefits of on-demand computing platforms to create a recommendation architecture that can adapt to variations in the number of model elements.

The first step in allowing the adoption of a cloud computing platform to address scalability issues is to analyze all steps taken to obtain the similarity between items. Basically this is a process where upon receiving information that a user has accessed an item, this user must be included in the corresponding item set, and then the similarity between this item and others must be recalculated using the algorithm described above to update the similarity graph. As the calculation of similarity between any two items is independent of the calculation of similarity between other pairs of items, this process can be executed concurrently. Therefore, it is possible to isolate problem components to address the scalability issue independently, thus creating a multi-layered architecture.

The first layer is responsible for receiving the information that a user has accessed a particular item. At this point, the engine receives the information that an identified user, for example, represented by an *UUID*, has viewed an item, also represented by an *UUID*. This information can be sent through a REST [26] interface, using an HTTP protocol through a call such as *http://:host:port/item/:UUID /user/:UUID*. The information can be temporarily stored in a queue, which will be consumed on demand, by the next processing layer, thereby isolating these layers and introducing control of information flow between them so that one does not overwhelm the other. Thus, this first layer can scale independently, adding or removing computing resources according to fluctuations in demand. This is a classic example of Infrastructure as a Service (IaaS) use, where, as more (fewer) users view items, more (fewer) HTTP requests regarding this information will be sent to the recommendation platform, and therefore, a greater (lesser) number of servers will be needed to handle these requests. Scalability control is performed automatically by monitoring agents that evaluate the server loads and create or destroy instances according to the fluctuating demand.

The second layer is composed of worker nodes that consume the *user:item* pairs from the queue and insert the user in the respective item set. Once consumed and stored, it is necessary to recalculate the similarity between the item whose set was modified, and the other existing items. Thus, in order to isolate the processing associated with this calculation to allow greater control over resources needed for the operation, this second layer only identifies which pairs of items need to have their similarity recalculated. In other words, the second layer consumes the *user:item* pairs output by the first layer, inserts the user into the respective item set, computes which pairs of items need to have their similarity recalculated, and finally places these in a queue. With this

approach, the second layer can be scaled as necessary so that its input queue should always be empty.

Finally, the third layer calculates the similarity between pairs of items consumed from the second layer's output queue, and updates the item's similarity graph. Figure 2, illustrates the described architecture.

One important optimization can be applied to *Layer II*, during the identification of item pairs that need to have their similarity recalculated. When using binary feedback, it is not necessary to recalculate the similarity between the item whose feedback is being processed and all other existing items. In this case, only those items already evaluated by the user and items that have a non-zero similarity with this one must be considered. For the items already evaluated by the user, the scalar product and magnitude will have changed. For the items that have a non-zero similarity with the item being processed, only the magnitude will have been updated, which will reduce the similarity. For the other items, since the scalar product and magnitudes remain unchanged, the cosine value does too.

Analyzing the architecture described above, there are still some critical points related to scalability. One is the data repository or storage of item sets, since the greater the number of items and amount of feedback are, the greater is the amount of data required to represent them. Another critical issue related to data access is the time required to read and write information, which can lead to request queuing as the number of nodes in the system increases. A third area of attention is the queues used to connect the different layers, which may limit the number of reads and writes, and cannot scale as the number of operations per second increases, for example, when there is a burst of user feedback.

To address problems related to the data repository, the proposed architecture uses a distributed in-memory representation, which isolates read operations from write operations. Existing solutions that use system memory to store data for faster access include Membase [27], Memcached [28], Hazelcast [29], and Redis [30]. We chose Redis as it comes with built-in data structures that are particular suited to the application in question: sets, sorted sets and lists. Additionally, Redis natively implements an intersection operation between sets, which can be used directly for the calculation of the dot product between the item vectors, as described above.

Another advantage of Redis is the existence of sorted sets. The representation of a similarity graph can be implemented using this structure, where each item corresponds to an ordered set of other items, with the sorting factor being the similarity between them. The use of Redis for similarity data storage allows the retrieval of similar items to be done in constant (fixed) time.

In addition, by using Redis as a data repository, it is possible to store the queues within it, as the implementation allows for the inclusion and exclusion of items in constant time. Redis holds up to 200,000 queue operations per second [30].

Finally, and more importantly, it is possible to set a life span for keys (sets, sorted sets, lists, etc.). This is particularly interesting for the recommendation engine, as it allows the inclusion of temporal dynamics in our solution, without the need to develop manual controls. In a nutshell, keys expire when their pre-determined lifespan is reached, and they are subsequently purged from the dataset. Thus, the introduction of temporal dynamics in our solution is achieved by setting an expiration time for each piece of feedback, which should reflect the period of time that the feedback is relevant to the recommendation model. After this expiration time, items are purged and no longer considered in the similarity calculation.

Figure 3 illustrates how Redis pools are integrated in the proposed architecture.

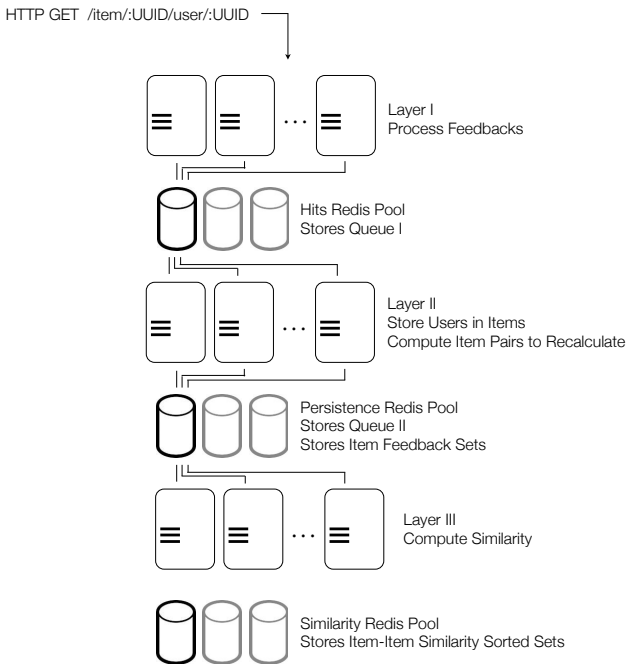

Figure 3. Redis pools connecting architecture layers.

As shown, besides the isolation of processing layers, there is also an insulation of data repositories, since each one has different requirements with respect to computing resources, amount of memory to represent the information, and the number of read/write operations. Thus, we can use the available resources of an IaaS platform more efficiently, which ultimately means obtaining the recommendation in less time and at a lower financial cost.

With the architecture layers isolated, it is possible to scale each one independently and on-demand. Thus, monitoring agents were developed to create or destroy server instances as needed. As mentioned, for the front-end (FE) servers that receive requests from users, resource consumption and response times are monitored to scale up or down consumption. In the other two layers, the monitor agent probes the processing queues size to create or destroy worker nodes with the aim of ensuring that queues are always empty, so that a recommendation can be obtained in real-time. To make this monitoring effective, these agents analyze the curve of the variation in queue size using a constant evaluation, which allows efficient elasticity.

Another important consideration is fault tolerance. With the layers isolated and the processing distributed, each node performs one operation at a time, initiated by the removal of an element from a queue, which takes place in constant time. Thus, the failure of any of the workers only means the loss of that work, which is not critical to the reliability of the recommendations made. Moreover, in the case of a node failure, the agents can start another one, as needed. Failures can easily be identified by the growth in the number of items in the queues. Regarding monitoring agent failover, these agents are executed in parallel on multiple servers and share information through a dedicated instance of Redis, also redundant, which ensures constant monitoring.

IV. DEPLOYMENT ON AWS

To validate the proposed architecture, it was deployed on the Amazon Web Services (AWS) cloud computing platform [3]. The first step in this process involved configuring the servers and their respective roles and creating images (Amazon Machine Images - AMIs) for each role to enable a rapid instance startup based on these images, according to demand fluctuations. Thus, images were created for: (1) the front-end servers, comprising a web server using Nginx with a Redis2 module, and able to write directly to the queue connecting the first two layers, (2) the nodes of the second layer, which consume the data in this first queue and structure the data based on the sets representing each item, (3) the nodes in the third layer, which calculate the similarity between items, (4) the servers that run the master and slave Redis instances, responsible for data storage, and (5) the servers that run the monitoring agents.

Once images had been created and servers configured, the load balancing service, Elastic Load Balancer (ELB), was employed to enable better scalability of user requests, and also to isolate access to data repositories, allowing the startup or shutdown of instances without any environment reconfiguration needed. Figure 4 shows the final architecture deployed on AWS.

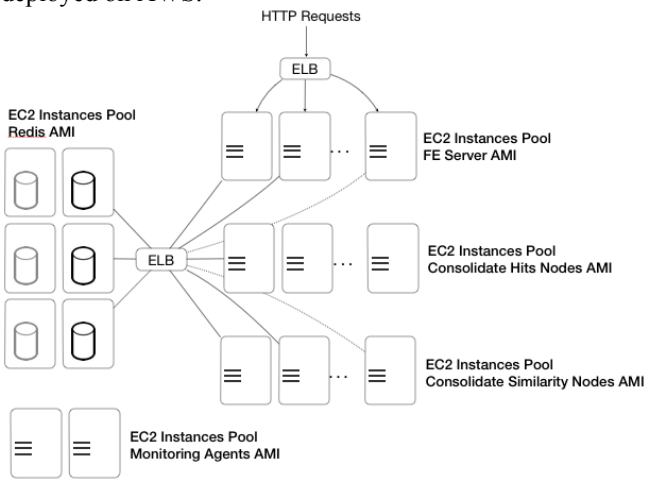

Figure 4. The recommendation platform deployed on AWS.

Note that any element of the architecture can be scaled up or down to meet demand, thus enabling complete scalability, which is crucial for dynamic environments with great variations in the numbers of users and items.

To validate the architecture, it was implemented in a real production scenario used for video recommendations on the Globo.com portal [15]. Globo.com is the Internet branch of Globo Organizations, the largest media group in Latin America, and the leader in the broadcasting media segment for Brazilian Internet audiences. As an Internet branch, Globo.com is responsible for supporting all companies in the group with their Internet initiatives. One of the most important of these is making available online all content produced for the Globo TV and PayTV channels. This means that Globo.com needs to produce more than 1000 new videos every day, and has more than 2 million videos available for its users. Moreover, this content is accessed by millions of unique users daily.

The integration of the Globo.com video platform (globo.tv) with the proposed recommendation engine was carried out such that for every video viewed, a request was made using the platform's REST interface stating that a particular user had watched a particular video, each identified by a UUID. A further point of integration was developed in the content offering, such that at the end screen of a video playback (Figure 5) and at player webpage (Figure 6), where suggested items are retrieved from the recommendation platform based on the item seen.

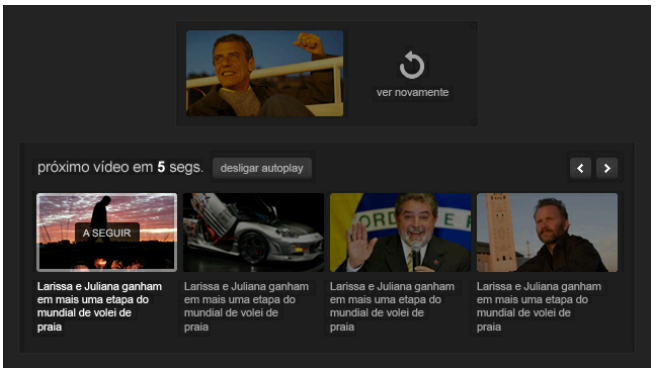

Figure 5. Recommendations integrated in the player and presented after playback.

Figure 6. Recommendations presented to user at globo.tv webpage.

A. Performance and Cost Analysis

In this integration, the platform was subjected to more than 10 million requests per day, with 300,000 different items and over 3 million unique users daily. On average, more than 1 billion similarity calculations were performed per day, since for each piece of feedback, 110 *item:item* pairs

were computed by *Layer II* with their similarity calculated by *Layer III*. Figure 7 shows the requests made during a ten day period by the player to the feedback interface, indicating that a user plays a video. Note that the volume of requests changes constantly, and that there are some request spikes, in this case, associated with real-time coverage of Brazilian Soccer League on the Globo.com website. It is exactly because of such demand fluctuations that a cloud computing platform is being used. In the test scenario, one Elastic Compute Cloud (EC2) [31] instance can handle all the requests for an average day; however, to support these high demand windows, it was necessary to use up to 4 FE instances, which were managed automatically by the monitoring agents.

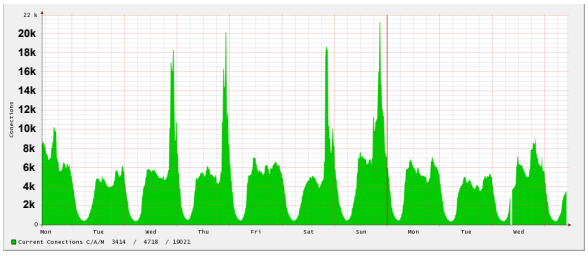

Figure 7. User feedback requests for a 10 day period starting on Monday.

Once received, the feedback (*user:item* pairs) was stored in *Queue I*, and then processed by *Layer II*. The auto-scaling algorithm, developed for the monitoring agents, was configured to probe the queue size every minute, and to automatically start a new instance if the queue size was greater than 1000 pairs. Once started, the instances where only terminated when the queue was completely empty. One important optimization that could be made concerns the cost management of this instance start/stop, process. Since Amazon's minimum charge is by the hour, it is not economical to terminate an instance with less than 1 hour of usage. This control could be implemented in the monitoring agents; however, for this test case, it was not used. With this configuration, 3 machines on average, were sufficient to store feedback and generate *item:item* pairs in *Layer II*. Figure 8 illustrates how the number of *Layer II* nodes changes (darker line – right axis) as the size of *Queue I* varies (lighter line – left axis) throughout a typical 24 hour period.

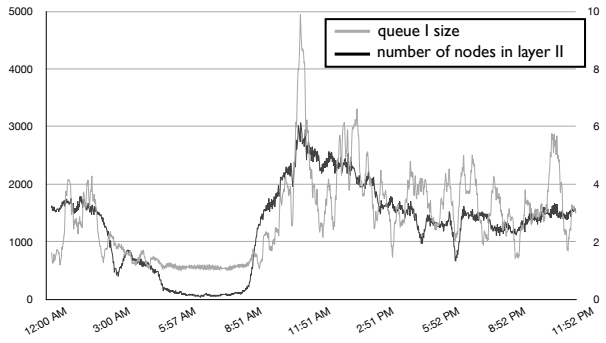

Figure 8. Variation in number of Layer II nodes according to Queue I size.

One important metric, which is directly associated with the real-time issue, is the time required to process the feedback and compute similarity pairs, since as the number of users and items grows, this time tends to increase, compromising how fast the similarity graph is updated. As shown in Figure 9, bellow, with distributed processing using automatic resource provisioning in the Cloud, the time required to process feedback and compute similarity pairs stabilizes at a constant value, around 23ms, although the number of unique users is linearly increasing.

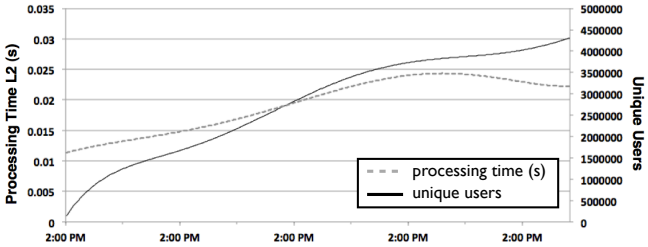

Figure 9. Processing Time in Layer II vs. Total Unique Users.

Regarding *Layer III*, the same scaling approach was adopted. Figure 10 illustrates how the number of *Layer III* nodes changes (darker line – right axis) as the size of *Queue II* varies (lighter line – left axis) on average.

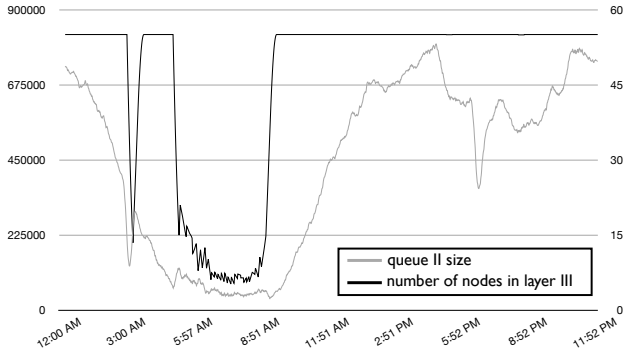

Figure 10. Variation in number of Layer III nodes according to Queue II size.

Note that the maximum number of *Layer III* nodes was limited to 55 nodes, because of project cost restrictions on Globo.com. However, this number could easily be changed in the agent configuration.

One important architecture component is the Redis pool, where all the data are stored. Since Redis is single processed, it could become a bottleneck as the number of instances reading and writing data increases. So, to distribute this load, a Master-Slave configuration was used, balancing writes to the Master and reads from the Slave. Furthermore, the three Redis roles were isolated on different processes, one for each role, to avoid competition between the different layers. In this scenario, each Redis instance has a different resource usage profile. The Hits Redis, which stores *Queue I*, can be fully stored in memory, without disk persistence, since it should has very fast queue IO, but, this queue must be kept empty most of the time. Its memory footprint is also very

low, since only *user:item* pairs are stored, and, the number of elements in the queue is limited.

The Persistence Redis, which stores the items sets and *Queue II*, has a completely different requirement. Since all information on user feedback is stored in this instance, it must store the data to disk, in order to recover the recommendation input data in case of Redis process failure. However, it should also have high IO rates, and thus, this data should be available in memory. As a result, the memory footprint is much higher than that of the Hits Redis, since it stores the information for all user feedback.

Finally, the Similarity Redis, which stores the similarity relations, has the same needs as the Persistence Redis, however, with a smaller memory requirement.

One important observation about Redis data persistence on disk is that it increases server disk IO, which could slow down the operations due to IO delays, and reduce the read and write throughput. So, to avoid this overhead with disk IO, one dedicated Slave was set up to perform data persistence. This Slave did not receive requests from other server instances, and its only responsibility was to dump data to disk. All other Redis servers were configured to use only memory as storage. As a result, in the Globo.com case, the Persistence Redis required 25 GB of memory to maintain all the data in memory, while the Similarity Redis used 20 GB.

Besides the scalability concerns, it is also important to analyze whether the proposed architecture is economically feasible. This cost analysis was carried out on the architecture deployed using the AWS Cloud platform.

For all tests performed, the EC2 large instance type was used to instantiate worker nodes, i.e., all servers used in layers I, II, and III. For the Redis pool, which is the data repository, EC2 Quadruple Extra Large instances were used, since these servers have high memory requirements. Furthermore, two ELB load balancers should be created to balance the load of the FE server and Redis pool. Amazon AWS also charges for data traffic, however, the data transfer costs are not significant compared to instance costs.

Finally, the total cost per month is approximately $15,000, with the details given below.

Nodes (50 on avg)	$12,240
Redis Servers (2 – master/slave)	$2,880
ELB	$ 36
Data Traffic, S3/EBS Storage, Elastic IPs	~ $100
Total per month	***$15,256***

V. CONCLUSIONS

In this paper we argued that the combination of mature research in collaborative filtering for item-item recommendations and emergent cloud computing technology opens up a great number of research challenges and possibilities. We introduced an architecture for real-time item-item recommendations, which rationalizes the use of resources by exploring on-demand computing. We also presented experimental results that demonstrate the feasibility of the proposed approach for environments with millions of users and items.

The fact that the elastic capacity of a public Cloud such as the AWS is very large, i.e., it offers access to practically unlimited resources, changes the fundamental premises that hitherto guided the research in this area and has the potential implications given below.

- The (recommendation quality x processing time) trade-off, no longer holds. New architectures can be developed aimed at maximum recommendation quality without input data sampling or model simplification, as processing times can be reduced by using parallel implementations such as the proposed approach.
- Choosing an optimal recommendation algorithm is known to be very difficult. The possibility of using several approaches, in parallel and at a low cost, may herald the beginning of a new era of experimentation in content suggestion.
- Adapting parallel approaches to work with linear algorithms requires a great deal of effort. In the case of the proposed approach, we developed a multi-layer architecture to allow for the use of independent scaling processes for each layer by addressing each processing layer independently. Although our results are encouraging, we acknowledge that much remains to be done. In particular, we believe that the development of self-tuning algorithms to determine optimal instance management could achieve very good results.

Finally we would like to highlight that the techniques used for the similarity calculations are flexible, i.e., they can be exchanged and customized as needed. This ensures flexibility, adaptation, extensibility, and generality of the proposed architecture.

REFERENCES

[1] Schafer, J. B., Konstan, J. and Riedl, J.: 1999, Recommender Systems in E- Commerce‘. In: EC ’99: Proceedings of the First ACM Conference on Electronic Commerce, Denver, CO, pp. 158-166.

[2] G. Linden, B. Smith, and J. York. Amazon.com recommendations: Item-to-item collaborative filtering. IEEE Internet Computing, 7(1):76-80, 2003.

[3] Amazon.com, http://www.amazon.com.

[4] Netflix, http://www.netflix.com.

[5] Funk, Simon. Netflix Update: Try This At Home. http://sifter.org/~simon/journal/20061211.

[6] Xavier Amatriain, Justin Basilico; “Netflix Recommendations: Beyond the 5 stars (Part 1)”, http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html

[7] Resnick P., and Varian H. R., “Recommender systems”, Communications of the ACM 40, 56–58, March 1997.

[8] Adomavicius G., Tuzhilin A. “Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions”, IEEE Transactions on Knowledge and Data Engineering 17, 734–749, June 2005.

[9] Xiaoyuan Su , Taghi M. Khoshgoftaar, A survey of collaborative filtering techniques, Advances in Artificial Intelligence, 2009, p.2-2.

[10] AnneY un-An Chen, Dennis McLeod, Collaborative Filtering for Information Recommendation Systems.

[11] J. Ben Schafer, Nathaniel Good, and Joseph Konstan et. al. Combining collaborative filtering with personal agents for better

recommendations. In Proceedings of the 1999 National Conference of the American Association of Artificial Intelligence, pages 439–436.

[12] J. L. Herlocker, J. A. Konstan, and J. Riedl. “Explaining collaborative filtering recommendations”, In Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work, ACM Press, pp. 241-250, 2000.

[13] I. Pilászy and D. Tikk, ‘Recommending new movies: even a few ratings are more valuable than metadata’, in Proc. of the third ACM conf. on Recommender systems, pp. 93–100, (2009). ACM.

[14] Lang K., Newsweeder: Learning to filter news, 12th International Conference on Machine Learning, pp. 331-339, 1995.

[15] globo.com, http://www.globo.com.

[16] Armbrust, M., Fox, A., Griffith, R., et al. (2009) “Above the Clouds: A Berkeley View of Cloud Computing”, In: University of California at Berkeley Technical Report no. UCB/EECS-2009-28, pp. 6-7, 2009

[17] Vogels, W., (2008) “A Head in the Clouds – The Power of Infrastructure as a Service”, In: First workshop on Cloud Computing in Applications (CCA’08), October, 2008.

[18] Vouk, M.A., Cloud Computing – Issues, Research and Implementations, 30th International Conference on Information Technology Interfaces, June, 2008, pp. 31-40

[19] Cearley, D. – Hype Cycle for Cloud Computing – 2009 – Gartner report number G00 168780, 2009.

[20] Billsus D., Pazzani M., “User Modeling for Adaptive News Access”. User-Modeling and User-Adapted Interaction 10(2-3), 147-180, 2000.

[21] L. Baltrunas, X. Amatriain, Towards Time-Dependant Recommendation based on Implicit Feedback. In Proceedings of the third ACM conference on Recommender systems, 2009, pp. 423-424.

[22] MovieLens, http://www.movielens.umn.edu.

[23] Yifan Hu , Yehuda Koren , Chris Volinsky, Collaborative Filtering for Implicit Feedback Datasets, Proceedings of the 2008 Eighth IEEE International Conference on Data Mining, p.263-272, 2008.

[24] Y. Koren, “Collaborative Filtering with Temporal Dynamics,” Proc. 15th ACM SIGKDD Int’l Conf. Knowledge Discovery and Data Mining (KDD 09), ACM Press, 2009, pp. 447-455.

[25] Souza, B.F.M. "Matrix Factorization Models For Video Recommendation". Unpublished master's thesis. PUC-Rio, 2011

[26] Zhang, L., RESTful Web Services. Web Services, Architecture Seminar, University of Helsink, 2004.

[27] Membase - http://www.couchbase.org/membase

[28] Memcached - http://memcached.org/

[29] Hazelcast - http://www.hazelcast.com/

[30] Redis, http://redis.io/.

[31] EC2 Elastic Compute Cloud (EC2) – http://aws.amazon.com/ec2/