7 Application in Globo.com and Results

To validate the proposed architecture, it was implemented in a real production scenario used for video recommendations on the Globo.com portal [18]. Globo.com is the Internet branch of Globo Group, which is 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 2000 new videos every day, and has more than 4 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 with the recommendation system was developed so as to offer users additional content at the end of the exhibition of any given video, as depicted by Figure 21. The top portion of the screen shows the last scene of the video on the left, and the replay button on the right (depicted as a curved arrow). The bottom of the screen contains a strip of video recommendations captioned by a small description of its contents.

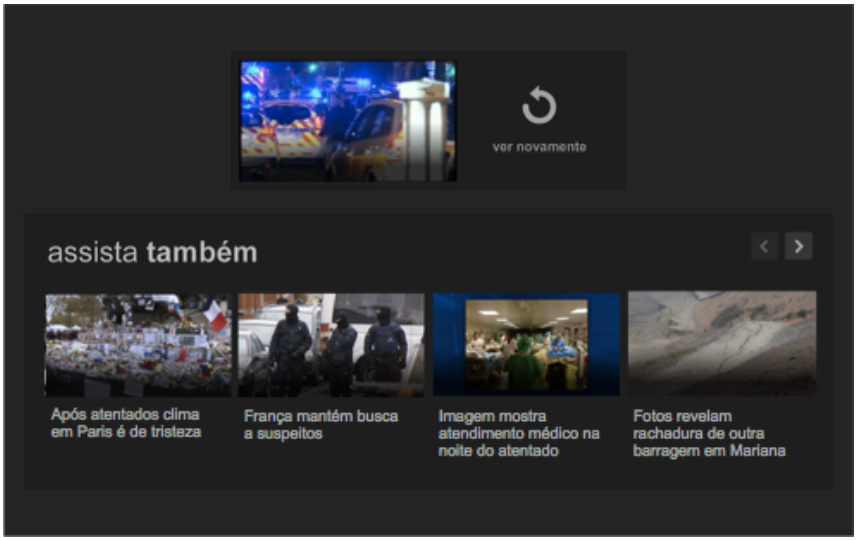

Figure 21 - Recommendations integrated in the player and presented at the end of the playback

Figure 22 - Recommendations integrated in the web page

In this integration, the platform was subjected to more than 15 million requests per day, with 500,000 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 23 shows the requests made during a ten days 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) [51] instance can handle all the requests for an average day; however, to support these high demand spikes, it was necessary to use up to 4 front

end instances, which were managed automatically by the monitoring agents and auto scale.

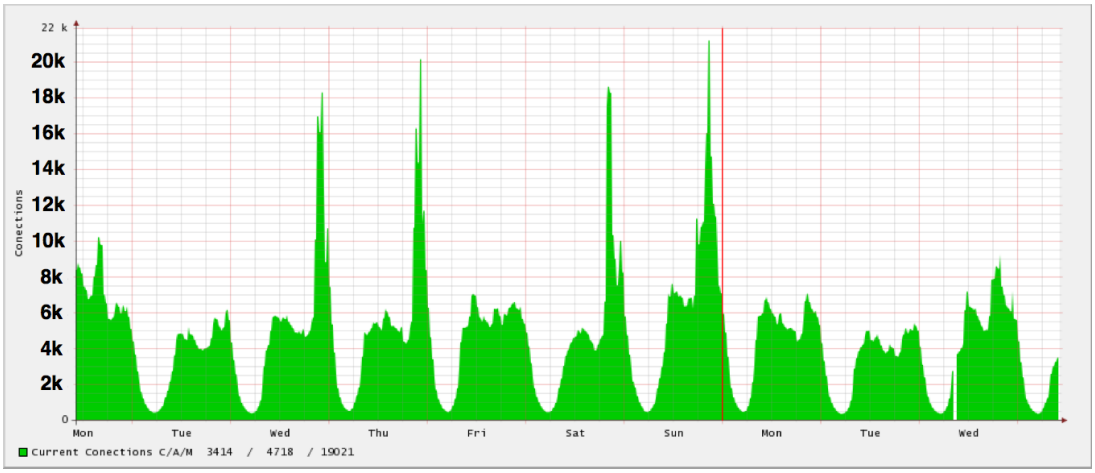

Figure 23 - User feedback requests for a 10 day period starting on Monday

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 24 illustrates how the number of Layer II nodes changes (darker-dashed line - right axis) as the size of Queue I varies (lighter line - left axis) throughout a 5 days period.

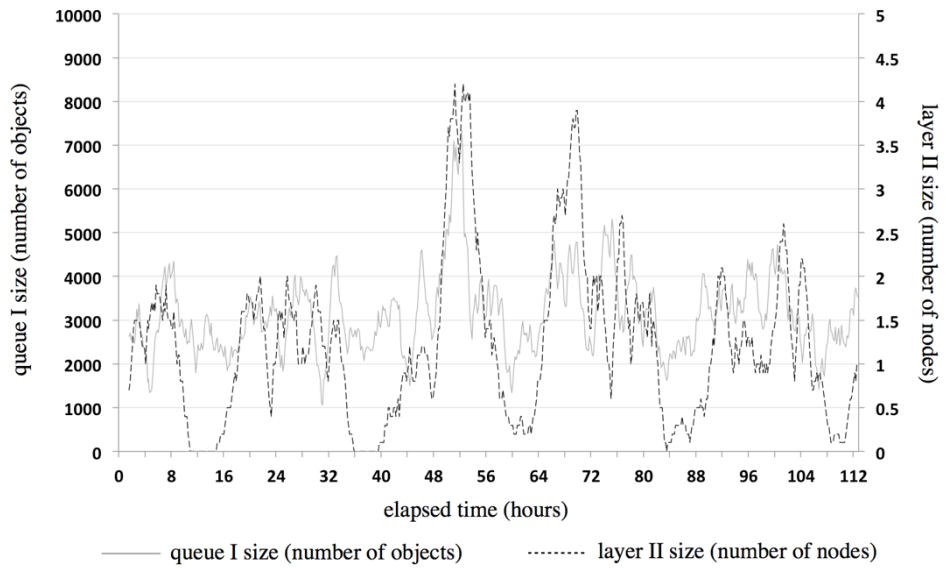

Figure 24 - Variation in number of Layer II nodes according to Queue I size

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 25, below, with distributed processing using automatic resource provisioning in the Cloud, the time required (dashed line – right axis) to process feedback and compute similarity pairs stabilizes at a constant value, around 23ms, although the number of unique users (continuous line – left axis) is increasing.

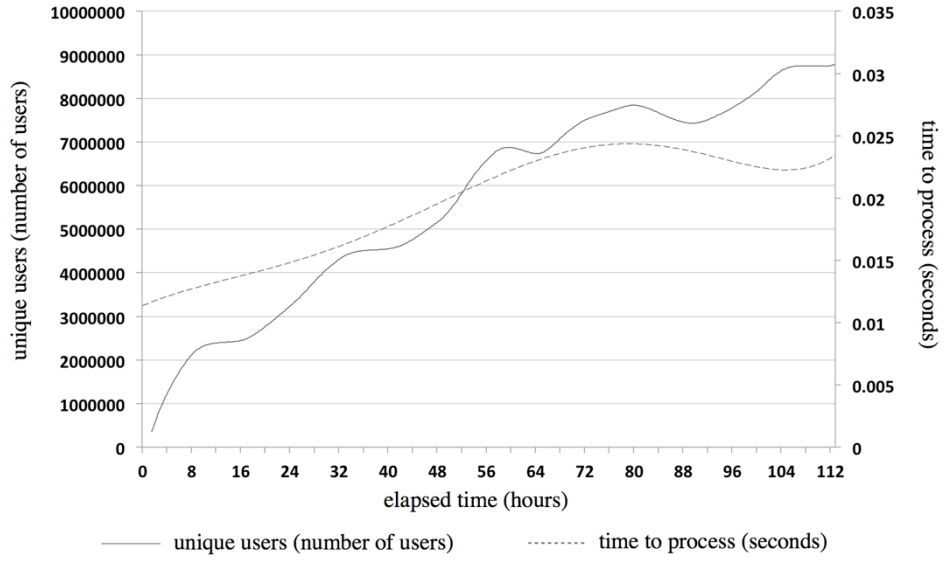

Figure 25 - Processing Time in Layer II vs. Total Unique Users

Regarding Layer III, the same scaling approach was adopted. Figure 26 illustrates how the number of Layer III nodes changes (dashed line – right axis) as the size of Queue II varies (continuous line – left axis) on average.

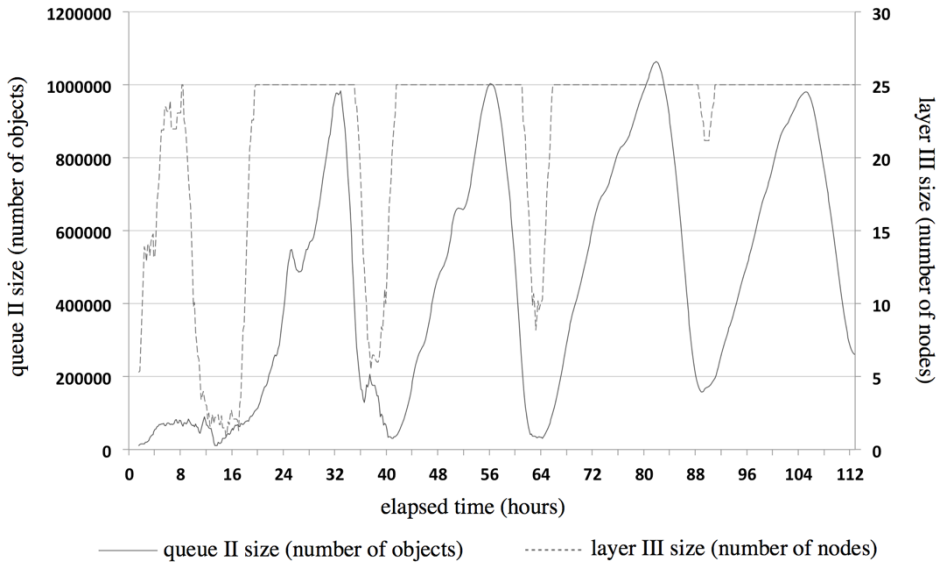

Figure 26 - Variation in number of Layer III nodes according to Queue II size

Note that the maximum number of Layer III nodes was limited to 25 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. In order to distribute this load, a Master-Slave configuration was used, balancing writes to the Master and reads from the Slave.

This Master-Slave configuration is native on Redis, and one master can have multiple slaves. Redis replication is non-blocking on the master side, this means that the master will continue to serve queries when one or more slaves perform the first synchronization. Replication is also non blocking on the slave side: while the slave is performing the first synchronization it can reply to queries using the old version of the data set. For synchronization, the master starts a background saving, and collects all new commands received that will modify the dataset. When the

background saving is complete, the master transfers the database file to the slave, which saves it on disk, and then loads it into memory. Then, the master will send to the slave all accumulated commands and all new commands received from clients that will modify the dataset. This is done as a stream of commands and is in the same format of the Redis protocol itself.

On our implementation, 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. In order 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. Figure 27 illustrates how the Redis instances are configured in the Persistence Redis Pool.

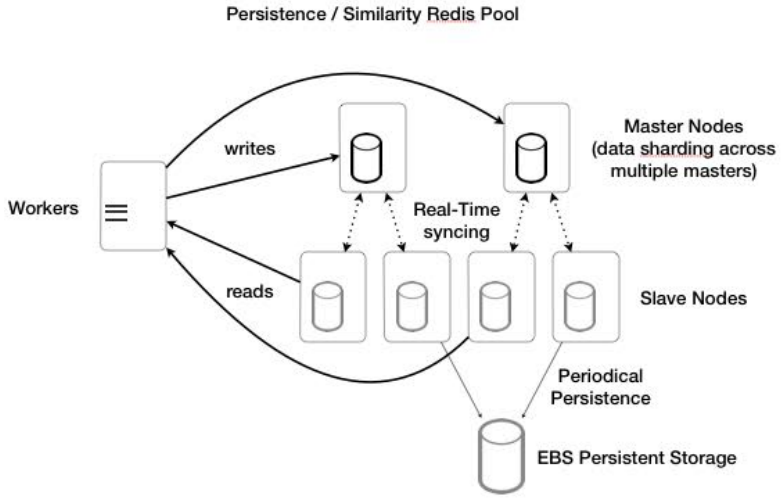

Figure 27 - Persistence Redis pool configuration

Basically, there are two master Redis instances that receive write commands from workers. Using Redis client, it is possible to distribute keys in several Redis instances using a sharding schema, where keys are uniformly distributed across instances to parcel out the load across several processes and machines. Each master Redis has two slaves, one dedicated for worker read operations and one dedicated for data persistence. Using this configuration, it is possible to scale read operations just by adding more read slaves. Furthermore, the data persistence IO operations are isolated in one dedicated slave, avoiding concurrency with read and write operations that are critical for real time similarity computation.

Besides the performance evaluation regarding scalability and usage of computational resources, another important aspect that was evaluated was the performance in terms of the quality of the produced recommendations, using the user conversion rate as metric, i.e. once displayed the recommendation, how many users actually clicked the recommended content. This is the key metric of efficiency of a recommendation system "who watched this video also watched these" [57].

For evaluation, a scale of five levels of feedback was chosen, using the adaptive implicit feedback model described previously. In addition, one type of binary feedback was tested, associated with the video. Tests were conducted for 30 days. The results can be seen in Figure 28.

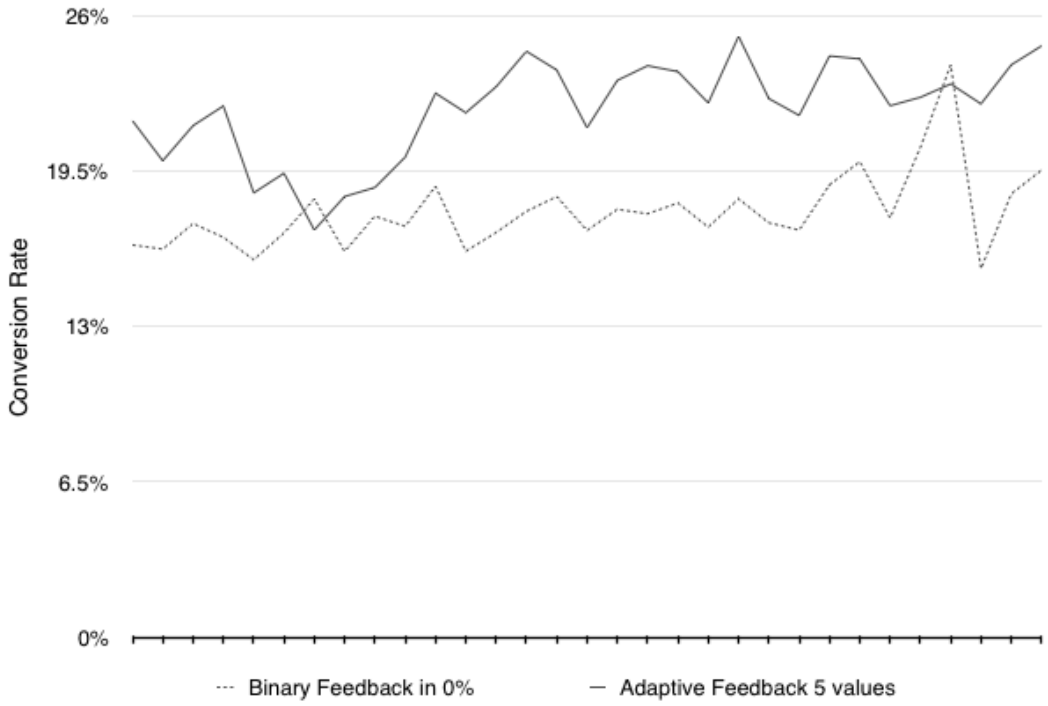

Figure 28 - Conversion rate comparison

As one can see, the generated recommendations from the proposed adaptive implicit feedback model obtained a higher conversion rate compared with the conversion rates when the binary feedback was used. Considering the average of 30 days, the conversion of the proposed model was 4.3% higher than the binary feedback associated with the video start. Using an offline collaborative filtering approach, with a daily batch processing and binary feedback, the conversion rate was 14% in average.

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.

Lastly, the total cost per month is approximately $9,000, with the details given below:

Nodes (25 on avg)		$6,120
Redis Servers (2 - master/slave)		$2,880
ELB		$36
Data Traffic, S3/EBS Storage, Elastic IPs	≈	$100
Total per month		$9,136

Table 2 - Total monthly cost for Globo.com recommendations deployment

This is the average cost per month to run the proposed real-time collaborative filtering architecture to produce recommendations, considering a scenario with millions of items, dozens of millions of unique users per month, and more than 500 million video views (user feedbacks). This shows that besides being scalable to very large datasets, and presents an interesting performance in conversion ratios, the proposed architecture also have a very low operating cost, considering the volume of data that is being processed, and compared to a more traditional approach that uses physical infrastructure.