4 A Real-Time Large Scale Collaborative Filtering Architecture

Based on the theoretical model for item-item similarity discussed in the previous section, we initially used a feedback matrix F 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. As also discussed in the previous section, 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, as also described by Pereira et. al. [70], 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 was 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] (a pseudocode is given in Algorithm 1). 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 [2].

```
for each existing item I1 do
    for each user U who performed a feedback for I1 do
        for each item I2 with a feedback from U do
            Record that a user sent a feedback for I1 and I2;
        end
    end
    for each item I2 do
        Compute the similarity between I1 and I2;
    end
end
```

Algorithm 1 - Similarity graph update [2]

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 indices that represents the 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 [59], i.e., the absence of feedback.

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's dimension *(M)*.

To exemplify the use of sets, suppose that there are $M = 10$ users, and that for the items I_1 and I_2 we have binary feedback vectors as follow: I_1 = (0,1,0,0,0,0,0,0,1,1) and I_2 = (0,1,0,0,1,0,1,0,1,0). So, for these two items their corresponding set is given by $Set(I_1)$ = {2, 9, 10} and $Set(I_2)$ = {2, 5, 7, 9}. The elements on these sets are the coordinates where the vectors I_1 and I_2 have a non-zero value. In order to calculate the similarity between I_1 and I_2, we use Equation 1:

$$Similarity(I_1, I_2) = \frac{\langle I_1, I_2 \rangle}{\|I_1\| . \|I_2\|} = \frac{\#\left(Set(I_1) \cap Set(I_2)\right)}{\sqrt{\#\left(Set(I_1)\right)} . \sqrt{\#\left(Set(I_2)\right)}} = \frac{2}{\sqrt{3} . \sqrt{4}},$$

where # represents the cardinality.

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.

4.1. A multi-layered architecture

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 in components to address the scalability issue independently, thus creating a multi-layered architecture. With a multi-layered architecture it is possible to address the scalability for each layer independently, thus increasing flexibility and resource usage efficiency.

The use of a multi-layered architecture is also interesting from a software engineering perspective, since it allows the flexibility to replace components as needed, reuse it individually in different scenarios or under different conditions. It also reduces dramatically maintenance and evolution costs, since each layer and/or component is responsible for one single task and has one well defined responsibility. Finally, a multi-layered approach is extremely interesting from a failure recover perspective, since it is easier to identify the point of failure and recover the work from that point, once each layer/component has one clear responsibility.

Figure 10 illustrates the proposed architecture, which will next be described.

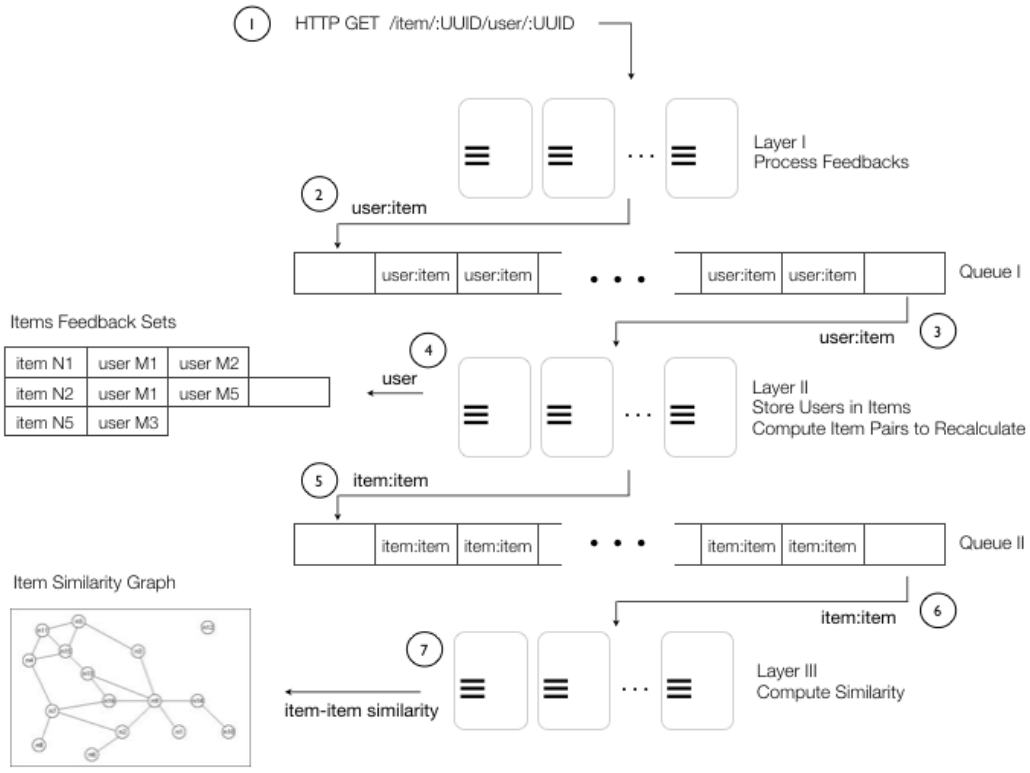

Figure 10 - Multi-layer collaborative filtering process

4.1.1. The first layer – Hits Layer

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 [60]

interface, using an HTTP protocol through a call such as *http://:host:port/item/:UUID/user/:UUID*.

This could be implemented in the player side, when in the first time that a user started playing a video, an unique identifier for that user is generated and stored in user's machine as a cookie. Then, for each video start, the player would send the information that the user started to watch a specific video through the HTTP call, as illustrated in the Figure 11.

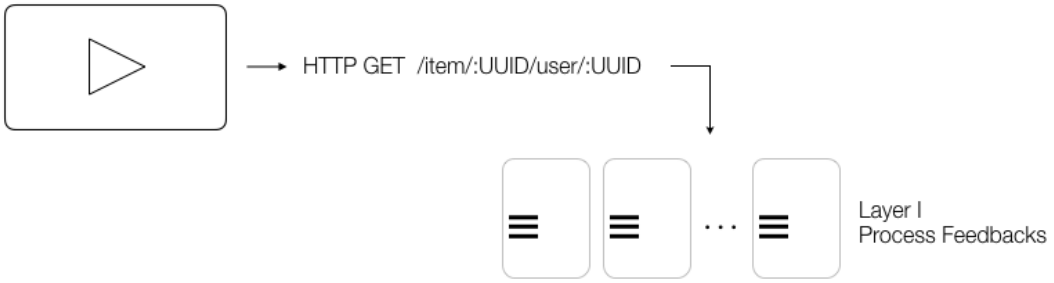

Figure 11 - Player sending the HTTP call informing that an user started to watch a video

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.

4.1.2. The second layer – Persistence Layer

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. For example, when the feedback from user U_3 to item I_2 is received, the U_3 identifier should be included in the set corresponding to I_2, which contains all user identifiers that already sent a

feedback regarding item I_2. Figure 12 below illustrates how the feedback information is stored in sets of items.

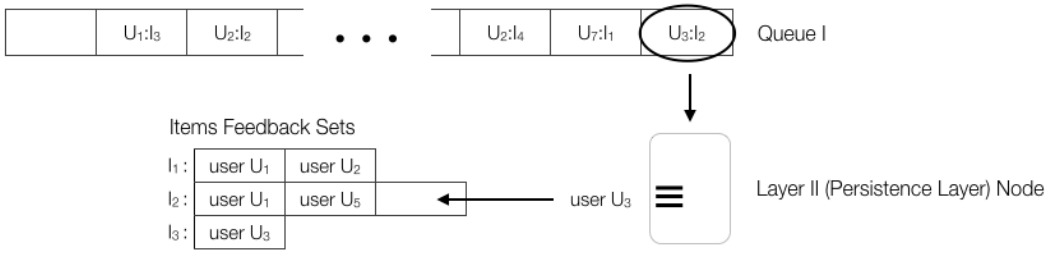

Figure 12 - Layer II storing binary feedback in sets

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. 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.

To allow a fast identification of the items already evaluated by the user that will need to have their similarities recalculated, one can use a supporting structure through sets, similar to that used to store user:item feedbacks. In this case, each user has a corresponding set containing the items already evaluated by him. Using the same example, when the feedback from user U_3 to item I_2 is received, the U_3 identifier should be included in the set corresponding to I_2, which contains all user identifiers that already sent a feedback regarding item I_2, and the identifier I_2 also should be included in the set corresponding to U_3, which contains all item identifiers that were already evaluated by the user U_3. With this supporting structure, to identify

which items were already evaluated by the user and need to have their similarity recalculated, it is only necessary to query the items inside the corresponding user set. The Figure 13 illustrates this process with the supporting structure.

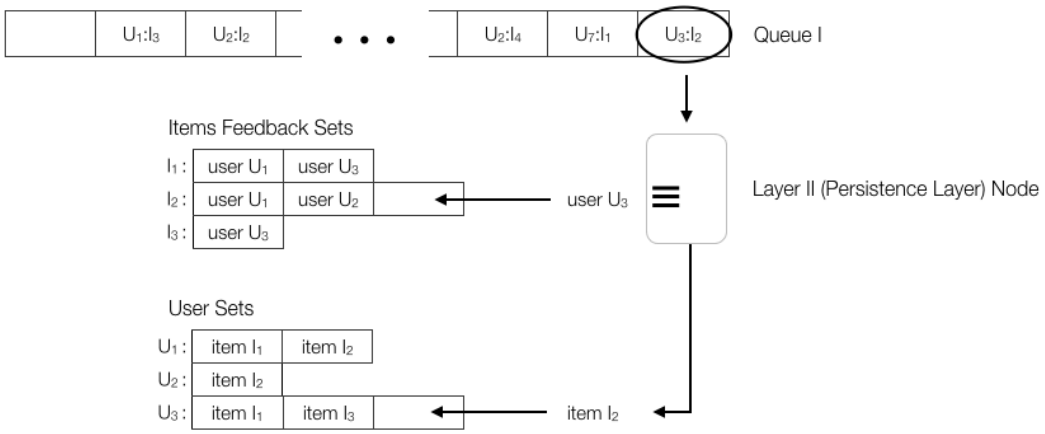

Figure 13 - Layer II storing items already evaluated by the user in a supporting set structure

Besides the items already evaluated by the user, it is also necessary to recalculate the similarity for the items that have a non-zero similarity with the item being processed. In this case, it is important to promptly identify which are those items, which should be done querying the similarity graph. So, store the similarity graph in an efficient way to allow for the fast querying of the items that have a non-zero similarity with the item being processed is an important requisite for the third layer, and will be detailed in the following.

Other important point to be considered by this layer is regarding temporal dynamics. As discussed previously, user's interests change over time, so, this should be considered in the recommender systems. This can be achieved from different ways, and one of them is discarding aged feedbacks. In this case, only recent feedbacks are considered. So, to introduce temporal dynamics in the proposed architecture, it is important to store a lifespan for each item and user set, so that they can be discarded after a specific period of time.

In summary, the second layer consumes the user:item pairs output by the first layer, inserts the user into the respective item set and the item in the respective user 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.

4.1.3. The third layer – Similarity Layer

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. As discussed previously, the similarity is computed through the cosine of the item vectors. Since only binary feedbacks are being considered, this computation can be done through the intersection of item sets, as stated above, and represented in the following equation:

$$Similarity(I_1, I_2) = \frac{\langle I_1, I_2 \rangle}{\|I_1\| . \|I_2\|} = \frac{\#(Set(I_1) \cap Set(I_2))}{\sqrt{\#(Set(I_1))} . \sqrt{\#(Set(I_2))}},$$

where # represents the cardinality.

So, since the layer II is structuring the feedbacks as item sets, to compute the similarity between two items the third layer just need to obtain the intersection between the sets representing those items and the size of each one of sets. Figure 14 below, illustrates the similarity calculation between two items, I_1 and I_2, through the intersection of sets stored by the layer II.

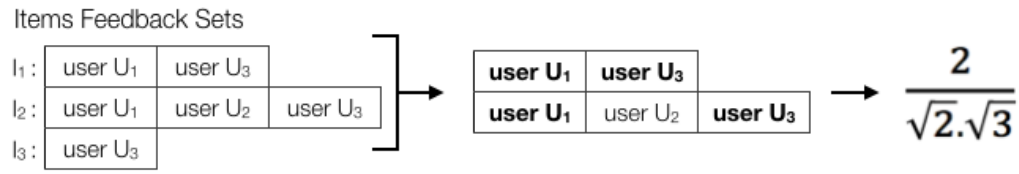

Figure 14 - Similarity calculation between items I_1 and I_2 through intersection of sets

The third layer can also be scaled as necessary, however, only a single worker node should operate over a specific pair of items at the same time, since the similarity calculation is not atomic. A lock mechanism using a shared key with timeout is used to avoid concurrency of multiple nodes over the same pair of items.

As described above, one important requisite of this third layer is the ability to store the similarity graph in a structure that allows a promptly identification of the items that have a non-zero similarity with the item being processed by the layer II.

Moreover, it is also important to store this graph using a scalable structure, that could handle millions of items with a minimum resource requirement.

In order to address those needs, a sorted set structure can be used to represent the item similarity graph. Basically, each item can be represented as a sorted set of items, ordered by their similarities. For example, consider that the similarity between the item I_2 and I_3 is 0.945, the similarity between I_2 and I_1 is 0.897, and the similarity between I_2 and I_4 is 0.602. In this case, this information could be store in a sorted set Set(I_2) = (I_3, I_1, I_4), being I_3 the I_2 most similar item and I_4 the less similar. Note that with this structure only the items that have non-zero similarity were stored, which allows for the fast identification of which pair of items should have their similarity recalculated due to change in magnitude, as discussed previously. Moreover, only essential information is stored, reducing the memory and storage requirements.

However, despite the scalability related to the number of user and items and their variations along the time, there is also an important point that the proposed architecture must address. It must be flexible and scalable enough to work with different implicit feedbacks rather than binary.

As mentioned in the beginning of this chapter, only a binary feedback is being used as implicit feedback until now, basically associated with the start watching action. However, this basic binary feedback may not consistently represent the actual user interest. So, it is important to analyze a different type of implicit feedback, and also how to use the proposed architecture with those different feedback types in order to keep all of its advantages.

The next chapter presents an alternative implicit feedback model that could be used with the proposed architecture, and that improves the quality of generated recommendations.