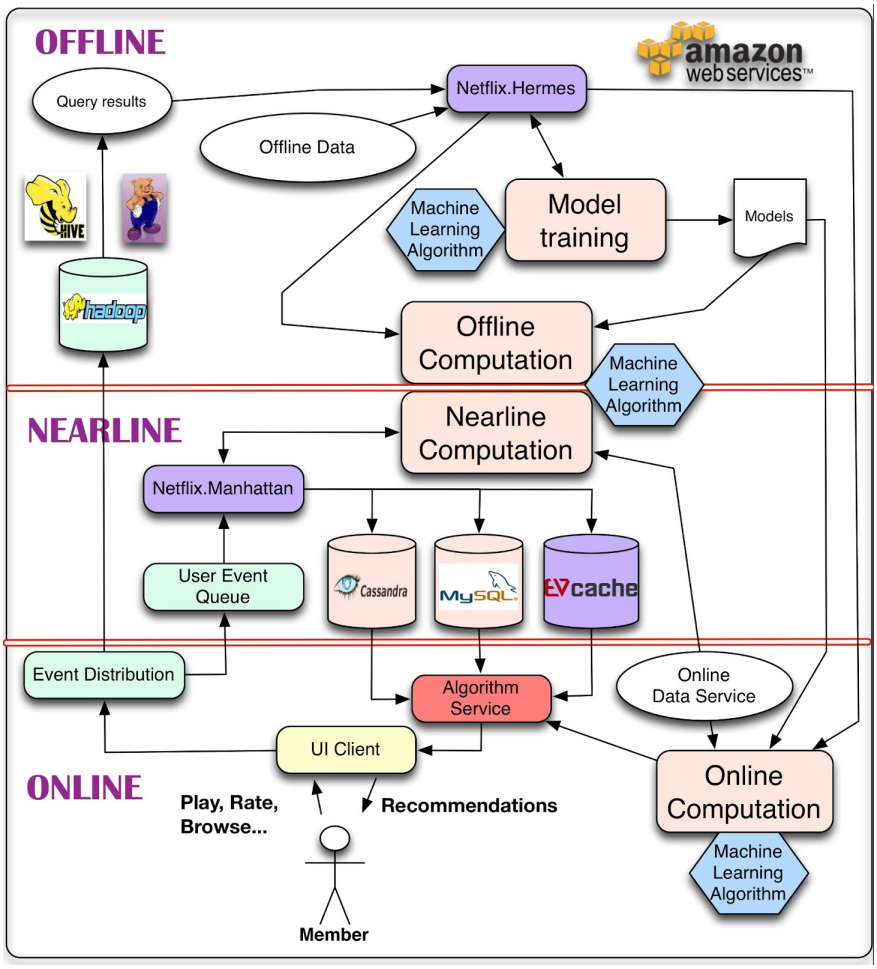
organizations
Mar 27, 2013 · 11 min read

System Architectures for Personalization and Recommendation

by Xavier Amatriain and Justin Basilico

In our previous posts about Netflix personalization, we highlighted the importance of using both data and algorithms to create the best possible experience for Netflix members. We also talked about the importance of enriching the interaction and engaging the user with the recommendation system. Today we're exploring another important piece of the puzzle: how to create a software architecture that can deliver this experience and support rapid innovation. Coming up with a software architecture that handles large volumes of existing data, is responsive to user interactions, and makes it easy to experiment with new recommendation approaches is not a trivial task. In this post we will describe how we address some of these challenges at Netflix.

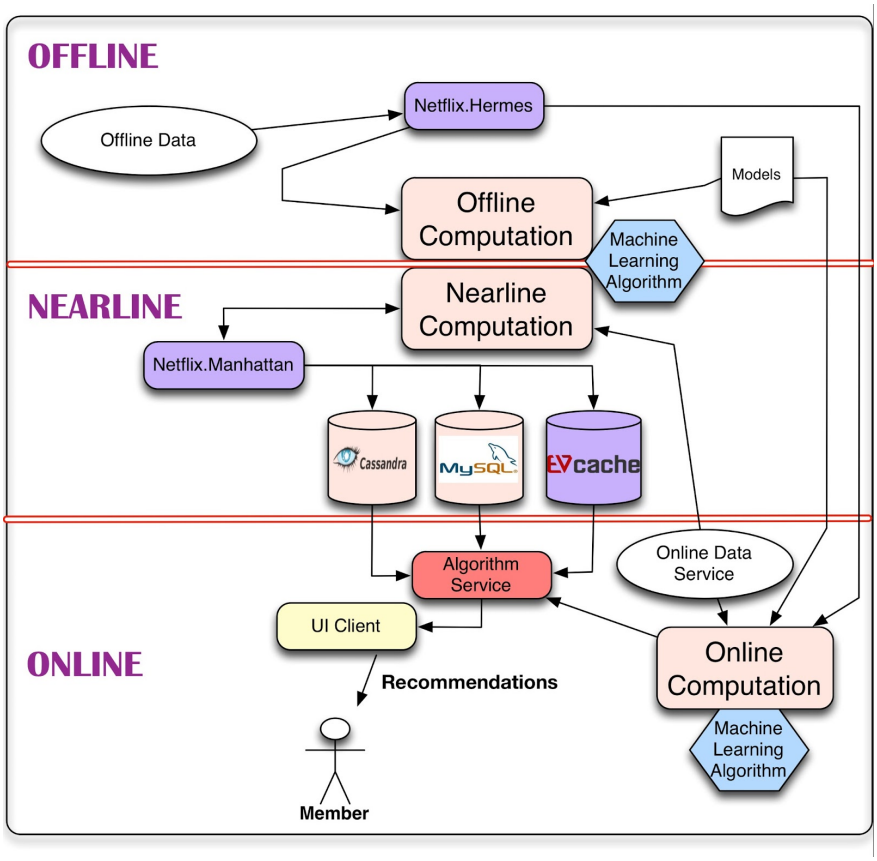
To start with, we present an overall system diagram for recommendation systems in the following figure. The main components of the architecture contain one or more machine learning algorithms.



The simplest thing we can do with data is to store it for later offline processing, which leads to part of the architecture for managing **Offline jobs**. However, computation can be done offline, nearline, or online. **Online computation** can respond better to recent events and user interaction, but has to respond to requests in real-time. This can limit the computational complexity of the algorithms employed as well as the amount of data that can be processed. **Offline computation** has less limitations on the amount of data and the computational complexity of the algorithms since it runs in a batch manner with relaxed timing requirements. However, it can easily grow stale between updates because the most recent data is not incorporated. One of the key issues in a personalization architecture is how to combine and manage online and offline computation in a seamless manner. **Nearline computation** is an intermediate compromise between these two modes in which we can perform online-like computations, but do not require them to be served in real-time. **Model training** is another form of computation that uses existing data to generate a model that will later be used during the actual computation of results. Another part of the architecture describes how the different kinds of events and data need to be handled by the **Event and Data Distribution** system. A related issue is how to combine the different **Signals and Models** that are

needed across the offline, nearline, and online regimes. Finally, we also need to figure out how to combine intermediate **Recommendation Results** in a way that makes sense for the user. The rest of this post will detail these components of this architecture as well as their interactions. In order to do so, we will break the general diagram into different sub-systems and we will go into the details of each of them. As you read on, it is worth keeping in mind that our whole infrastructure runs across the public Amazon Web Services cloud.

Offline, Nearline, and Online Computation



As mentioned above, our algorithmic results can be computed either online in real-time, offline in batch, or nearline in between. Each approach has its advantages and disadvantages, which need to be taken into account for each use case.

Online computation can respond quickly to events and use the most recent data. An example is to assemble a gallery of action movies sorted for the member using the current context. Online components are subject to an availability and response time Service Level Agreements (SLA) that specifies the maximum latency of the process in responding to requests from client applications while our member is waiting for

recommendations to appear. This can make it harder to fit complex and computationally costly algorithms in this approach. Also, a purely online computation may fail to meet its SLA in some circumstances, so it is always important to think of a fast fallback mechanism such as reverting to a precomputed result. Computing online also means that the various data sources involved also need to be available online, which can require additional infrastructure.

On the other end of the spectrum, offline computation allows for more choices in algorithmic approach such as complex algorithms and less limitations on the amount of data that is used. A trivial example might be to periodically aggregate statistics from millions of movie play events to compile baseline popularity metrics for recommendations. Offline systems also have simpler engineering requirements. For example, relaxed response time SLAs imposed by clients can be easily met. New algorithms can be deployed in production without the need to put too much effort into performance tuning. This flexibility supports agile innovation. At Netflix we take advantage of this to support rapid experimentation: if a new experimental algorithm is slower to execute, we can choose to simply deploy more Amazon EC2 instances to achieve the throughput required to run the experiment, instead of spending valuable engineering time optimizing performance for an algorithm that may prove to be of little business value. However, because offline processing does not have strong latency requirements, it will not react quickly to changes in context or new data. Ultimately, this can lead to staleness that may degrade the member experience. Offline computation also requires having infrastructure for storing, computing, and accessing large sets of precomputed results.

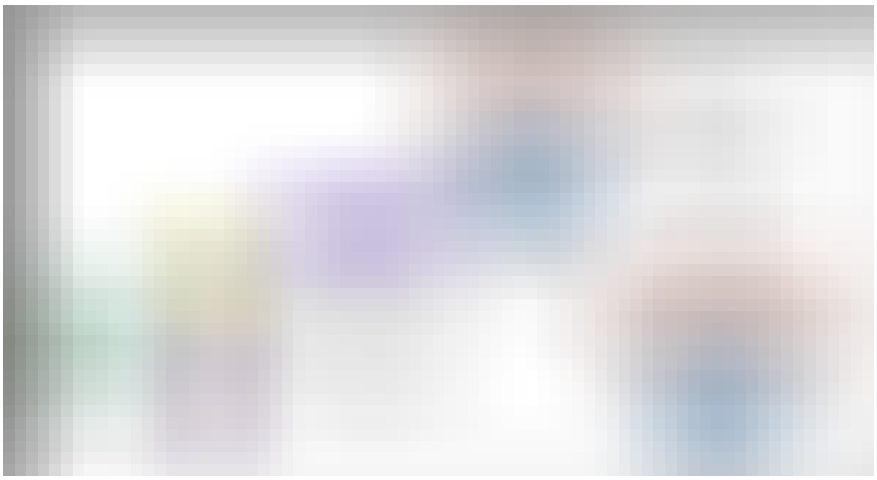
Nearline computation can be seen as a compromise between the two previous modes. In this case, computation is performed exactly like in the online case. However, we remove the requirement to serve results as soon as they are computed and can instead store them, allowing it to be asynchronous. The nearline computation is done in response to user events so that the system can be more responsive between requests. This opens the door for potentially more complex processing to be done per event. An example is to update recommendations to reflect that a movie has been watched immediately after a member begins to watch it. Results can be stored in an intermediate caching or storage back-end. Nearline computation is also a natural setting for applying incremental learning algorithms.

In any case, the choice of online/nearline/offline processing is not an either/or question. All approaches can and should be combined. There

are many ways to combine them. We already mentioned the idea of using offline computation as a fallback. Another option is to precompute part of a result with an offline process and leave the less costly or more context-sensitive parts of the algorithms for online computation.

Even the modeling part can be done in a hybrid offline/online manner. This is not a natural fit for traditional supervised classification applications where the classifier has to be trained in batch from labeled data and will only be applied online to classify new inputs. However, approaches such as Matrix Factorization are a more natural fit for hybrid online/offline modeling: some factors can be precomputed offline while others can be updated in real-time to create a more fresh result. Other unsupervised approaches such as clustering also allow for offline computation of the cluster centers and online assignment of clusters. These examples point to the possibility of separating our model training into a large-scale and potentially complex global model training on the one hand and a lighter user-specific model training or updating phase that can be performed online.

Offline Jobs

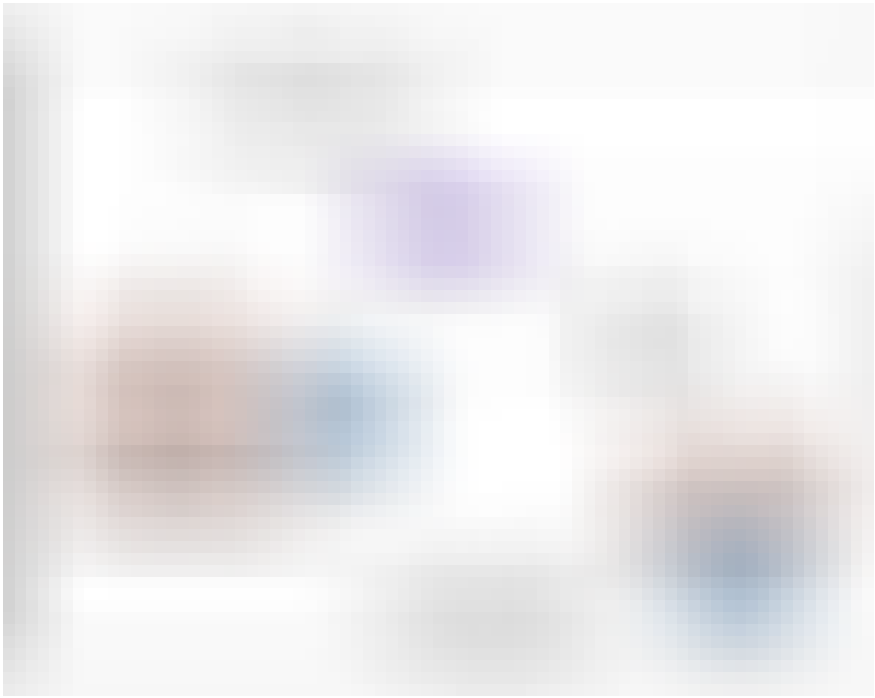


Much of the computation we need to do when running personalization machine learning algorithms can be done offline. This means that the jobs can be scheduled to be executed periodically and their execution does not need to be synchronous with the request or presentation of the results. There are two main kinds of tasks that fall in this category: **model training** and **batch computation of intermediate or final results**. In the model training jobs, we collect relevant existing data and apply a machine learning algorithm produces a set of model parameters (which we will henceforth refer to as the model). This

model will usually be encoded and stored in a file for later consumption. Although most of the models are trained offline in batch mode, we also have some online learning techniques where incremental training is indeed performed online. Batch computation of results is the offline computation process defined above in which we use existing models and corresponding input data to compute results that will be used at a later time either for subsequent online processing or direct presentation to the user.

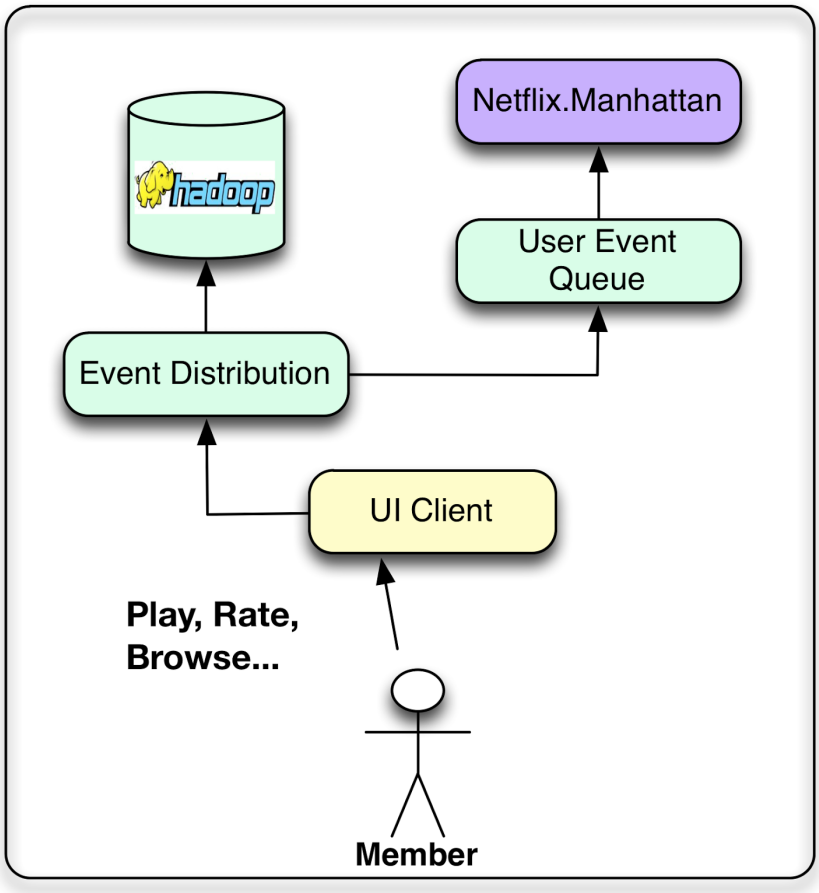
Both of these tasks need refined data to process, which usually is generated by running a database query. Since these queries run over large amounts of data, it can be beneficial to run them in a distributed fashion, which makes them very good candidates for running on Hadoop via either Hive or Pig jobs. Once the queries have completed, we need a mechanism for publishing the resulting data. We have several requirements for that mechanism: First, it should notify subscribers when the result of a query is ready. Second, it should support different repositories (not only HDFS, but also S3 or Cassandra, for instance). Finally, it should transparently handle errors, allow for monitoring, and alerting. At Netflix we use an internal tool named **Hermes** that provides all of these capabilities and integrates them into a coherent publish-subscribe framework. It allows data to be delivered to subscribers in near real-time. In some sense, it covers some of the same use cases as Apache Kafka, but it is not a message/event queue system.

Signals & Models



Regardless of whether we are doing an online or offline computation, we need to think about how an algorithm will handle three kinds of inputs: models, data, and signals. Models are usually small files of parameters that have been previously trained offline. Data is previously processed information that has been stored in some sort of database, such as movie metadata or popularity. We use the term “signals” to refer to fresh information we input to algorithms. This data is obtained from live services and can be made of user-related information, such as what the member has watched recently, or context data such as session, device, date, or time.

Event & Data Distribution

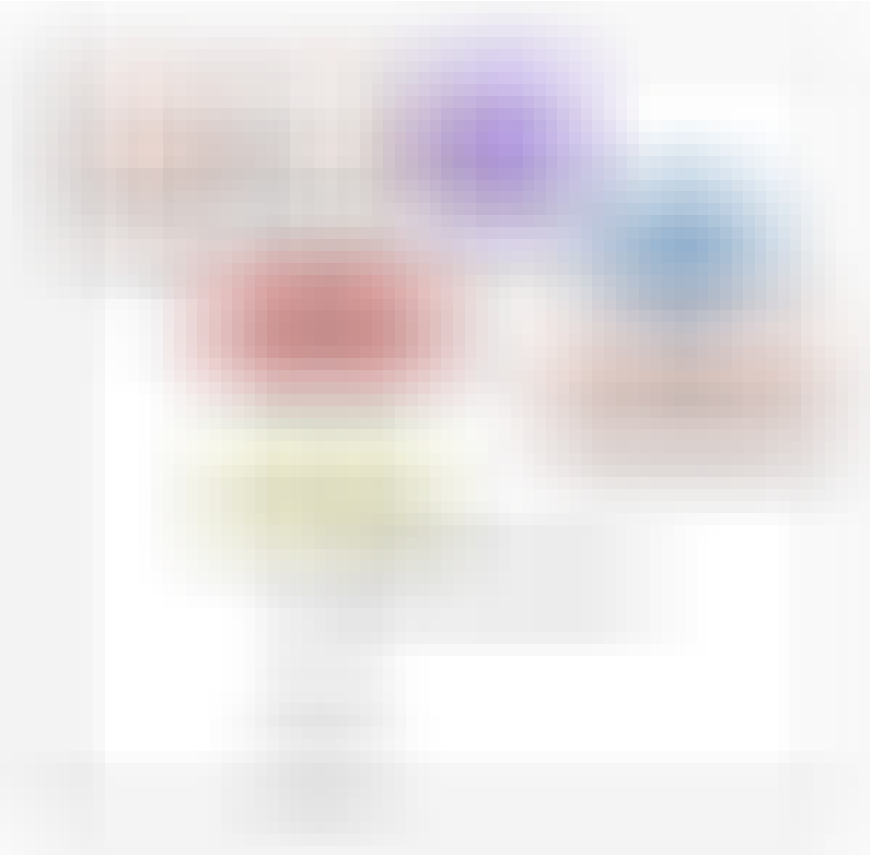


Our goal is to turn member interaction data into insights that can be used to improve the member's experience. For that reason, we would like the various Netflix user interface applications (Smart TVs, tablets, game consoles, etc.) to not only deliver a delightful user experience but also collect as many user events as possible. These actions can be related to clicks, browsing, viewing, or even the content of the viewport at any time. Events can then be aggregated to provide base data for our algorithms. Here we try to make a distinction between **data** and **events**, although the boundary is certainly blurry. We think of events as small units of time-sensitive information that need to be processed with the least amount of latency possible. These events are routed to trigger a subsequent action or process, such as updating a nearline result set. On the other hand, we think of data as more dense information units that might need to be processed and stored for later use. Here the latency is not as important as the information quality and quantity. Of course, there are user events that can be treated as both events and data and therefore sent to both flows.

At Netflix, our near-real-time event flow is managed through an internal framework called **Manhattan**. Manhattan is a distributed computation system that is central to our algorithmic architecture for

recommendation. It is somewhat similar to Twitter's Storm, but it addresses different concerns and responds to a different set of internal requirements. The data flow is managed mostly through logging through Chukwa to Hadoop for the initial steps of the process. Later we use Hermes as our publish-subscribe mechanism.

Recommendation Results



The goal of our machine learning approach is to come up with personalized recommendations. These recommendation results can be serviced directly from lists that we have previously computed or they can be generated on the fly by online algorithms. Of course, we can think of using a combination of both where the bulk of the recommendations are computed offline and we add some freshness by post-processing the lists with online algorithms that use real-time signals.

At Netflix, we store offline and intermediate results in various repositories to be later consumed at request time: the primary data stores we use are Cassandra, EVCache, and MySQL. Each solution has advantages and disadvantages over the others. MySQL allows for storage of structured relational data that might be required for some

future process through general-purpose querying. However, the generality comes at the cost of scalability issues in distributed environments. Cassandra and EVCache both offer the advantages of key-value stores. Cassandra is a well-known and standard solution when in need of a distributed and scalable no-SQL store. Cassandra works well in some situations, however in cases where we need intensive and constant write operations we find EVCache to be a better fit. The key issue, however, is not so much where to store them as to how to handle the requirements in a way that conflicting goals such as query complexity, read/write latency, and transactional consistency meet at an optimal point for each use case.

Conclusions

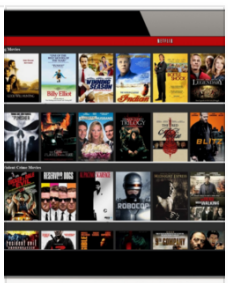
In previous posts, we have highlighted the importance of data, models, and user interfaces for creating a world-class recommendation system. When building such a system it is critical to also think of the software architecture in which it will be deployed. We want the ability to use sophisticated machine learning algorithms that can grow to arbitrary complexity and can deal with huge amounts of data. We also want an architecture that allows for flexible and agile innovation where new approaches can be developed and plugged-in easily. Plus, we want our recommendation results to be fresh and respond quickly to new data and user actions. Finding the sweet spot between these desires is not trivial: it requires a thoughtful analysis of requirements, careful selection of technologies, and a strategic decomposition of recommendation algorithms to achieve the best outcomes for our members. We are always looking for great engineers to join our team. If you think you can help us, be sure to look at our jobs page.

See Also:

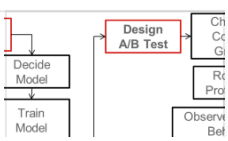
Netflix Recommendations: Beyond the 5 stars (Part 1)

One of the most valued Netflix assets is our recommendation system

medium.com

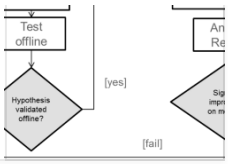


Netflix Recommendations: Beyond the 5 stars (Part 2)



More insight into our broader personalization technology.

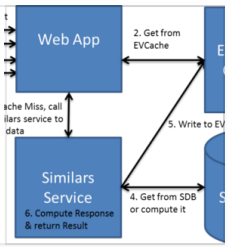
medium.com



Ephemeral Volatile Caching in the cloud

a highly scalable memcache-based caching solution

medium.com



. . .

Originally published at techblog.netflix.com on March 27, 2013.