TencentRec: Real-time Stream Recommendation in Practice

Yanxiang Huang[♯] Bin Cui[♯] Wenyu Zhang[§] Jie Jiang[§] Ying Xu[♯]
[♯]Key Lab of High Confidence Software Technologies (MOE), School of EECS, Peking University
[§]Tencent Inc.
[♯]{yx.huang, bin.cui, lauraxu1202}@pku.edu.cn
[§]{gabyzhang,zeus}@tencent.com

ABSTRACT

With the arrival of the big data era, opportunities as well as challenges arise in both industry and academia. As an important service in most web applications, accurate real-time recommendation in the context of big data is of high demand. Traditional recommender systems that analyze data and update models at regular time intervals cannot satisfy the requirements of modern web applications, calling for real-time recommender systems.

In this paper, we tackle the “big", “real-time" and “accurate" challenges in real-time recommendation, and propose a general real-time stream recommender system built on Storm named TencentRec from three aspects, i.e., “system", “algorithm", and “data". We analyze the large amount of data streams from a wide range of applications leveraging the considerable computation ability of Storm, together with a data access component and a data storage component developed by us. To deal with various application specific demands, we have implemented several classic practical recommendation algorithms in TencentRec, including the item-based collaborative filtering, the content based, and the demographic based algorithms. Specially, we present a practical scalable item-based CF algorithm in detail, with the super characteristics such as robust to the implicit feedback problem, incremental update and real-time pruning. With the enhancement of real-time data collection and processing, we can capture the recommendation changes in real-time. We deploy the TencentRec in a series of production applications, and observe the superiority of TencentRec in providing accurate real-time recommendations for 10 billion user requests everyday.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Data mining*

General Terms

Algorithms, Performance, Design, Experimentation, Reliability

Keywords

Real-time Recommendation, Scalability, Big Data, Practice, Application

1. INTRODUCTION

The big data era has provided users with rich information and influenced more and more people’s daily life. At the same time, it brings great opportunities and challenges to many fields from industry to research [6]. How to extract useful knowledge from the massive data becomes a crucial issue and attracts increasing attention on it. Recommender systems help solve the information overload problem by providing users with personalized service utilizing data mining techniques, and have been applied into many industrial fields. Meanwhile, recommender systems have attracted many researchers’ attention. Research about recommender systems has become an important research field nowadays [1].

Traditional recommender systems that analyze data and update models at regular time intervals, e.g., hours or days, cannot meet the real-time demands. For example, if a user posts a tweet “I’d like to watch a movie.", traditional recommender systems that update model after hours or days will miss this instant demand. Take advertising as another example, nowadays advertisements usually have very short life cycles, such as the advertisement for flash sale which is only active for several minutes. Traditional recommender systems cannot make fast responses to users’ preference changes and capture the users’ real-time interests, thus resulting in bad recommendation results. Since users’ real-time demands usually fade away as time goes on, traditional recommender systems’ prediction quality drops in the scenario where users’ preferences change fast.

Real-time recommender systems [9, 3] show their superiority over traditional ones in many cases. Unlike traditional ones, real-time recommender systems update more frequently, able to catch users’ instant need with very short delay like several seconds or milliseconds. However, current real-time recommender systems have their limitations. They either lack of the scalability in handing massive data, or are unable to produce accurate real-time recommendations in practice because of the problems in real-world such as implicit feedback problem and data sparsity problem. This leads to the high demand for practical real-time recommender systems in the industrial community.

The big data era poses great challenges on real-time recommender systems. The recommender system needs to satisfy multiple requirements. Consider a query that “During last ten seconds, what is the CTR (Click Through Rate) of an advertisement among the male users in Beijing, whose age is from twenty to thirty". It is a common query in current recommender systems, and has the following characteristics. First, it requires real-time response, usually

SIGMOD’15, May 31–June 4, 2015, Melbourne, Victoria, Australia.
Copyright © 2015 ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2742785.

in milliseconds. Second, it needs high computational complexity. Take the above query as example, we need to compute the combination of four dimensions including region, age, gender, and advertisement, leading to a large amount of calculation in the big data era. Third, as the result evolves over time, we need to capture users' real-time demands to provide accurate recommendations. In this case, the traditional batch processing techniques cannot satisfy the requirements. We need a recommender system that can execute large amount of real-time processing.

In this paper, we aim to provide data-driven, real-time, accurate, personalized recommendations for a wide range of production applications in Tencent Inc., and propose a general real-time recommender system, named TencentRec. We first discuss the salient requirements of real-time recommendation, which to a large extent dictate the design of TencentRec. The requirements of real-time recommendation can be described by three keywords, i.e., "big", "real-time", "accurate". First of all, the recommender system should be able to handle the massive data, which is described by "big". Second, the system must be real-time. On one side, it needs to response to users' queries in real-time. On the other side, it should be able to capture users' interests in real-time. Last, the "accurate" indicates that the recommender system should recommend the right items to users in order to improve the users' satisfaction and increase the earnings. These requirements are not isolated but are connected and influenced each other.

To tackle above requirements, we design TencentRec from three points of view, that is "system", "algorithm", and "data". We select Storm [1] to support the real-time computations in TencentRec, leveraging its computation ability to analyze the large amount of data streams. To handle the various data access and massive status data storage, we develop two components respectively named Tencent Data Access (TDAccess) and Tencent Data Store (TDStore). Since the situations of various production applications are different, it is almost impossible to build a one-size-fit-all algorithm to facilitate the different requirements of a wide range of applications. To satisfy diverse application requirements, we have implemented a series of classic recommendation algorithms, such as item-based collaborative filtering [16], demographic based algorithm [19] and the content based algorithm [18] on the Storm. We design several practical mechanisms for applying real-time recommendation algorithms fit for the scale of the distributed recommender system and retaining their effectiveness to produce accurate recommendations in real-time, including the data sparsity solution employing demographic methods, and the real-time filtering techniques. Specially, we propose a practical scalable item-based collaborative filtering algorithm to achieve the super characteristics such as robust to the implicit feedback problem, scalable incremental update and real-time pruning.

By collecting and processing the data in real-time, we can capture the users' preferences and interests changes in real-time. Whenever an event occurs, it costs less than one second for TencentRec to respond to this change and update the recommendation results. TencentRec is already in production usage and has been applied to a series of applications. It deals with 10 billion user requests and more than 1 PB data per day, and achieves superior performance in providing accurate real-time recommendations to users.

In summary, this paper makes the following contributions:

- We implement a general comprehensive real-time recommender system named TencentRec with a series of classic recommendation algorithms, which can handle large volume of data and provide accurate real-time recommendations for a wide range of production applications.
- We develop several practical mechanisms for applying real-time recommendation algorithms in practice to produce accurate recommendations. Specially, a scalable incremental item-based collaborative filtering algorithm is presented in detail.
- With elaborate deployment, TencentRec is currently used in a series of production applications in Tencent Inc. and improves the applications' performance significantly, demonstrating the superiority of TencentRec.

The rest of the paper is organized as follows. In Section 2, we review the related work. We introduce the framework of TencentRec in Section 3. Section 4 describes the practical scalable algorithms we proposed. And the implementation details are presented in Section 5. In Section 6, we demonstrate the production usage of TencentRec in applications and show its performance. Finally, we conclude our work in Section 7.

2. RELATED WORK

Recommender system has become an important research area since the mid-1990s [11, 22, 28]. There has been much work done in both industry and academia. The classic recommendation methods can be categorized into collaborative filtering and content based recommendations [1]. Collaborative filtering [16, 5] aims to recommend the items similar to the ones the users preferred in the past while the content-based methods [18, 23] recommend the items that are similar to the users' preferences learned. However, each recommendation method has its limitations, like the "cold-start" problem (new user problem and new item problem) and the data sparsity problem. As a result, a large amount of hybrid recommendation approaches aimed to take the advantages of both collaborative and content-based systems to avoid certain limitations [17, 19, 20, 27, 15]. Different ways were deployed to combine collaborative and content-based methods, including combing separate predictions, incorporating one's characteristics into another and building a general unifying model [1]. Moreover, topic model was utilized to do recommendations in [5, 13, 14] to discover the users' intrinsic interests. Other works [34, 33, 31, 32] adopt topic models to take other factors into recommendation, such as the temporal influences and the social behaviors.

Recently, real-time recommender system has attracted growing attention [21, 2]. Das et al. addressed the large data and fast changing content problem in recommender systems, and proposed online models to generate recommendations for users of Google News [7]. In [9] and [4], the authors provided users with real-time topic recommendations by analyzing social streams. However, their limitation is the scalability in handing the large amount of data. StreamRec was proposed in [3] which implemented a scalable collaborative filtering recommender model based on a stream processing system. However, it requires the explicit feedback data, i.e., user ratings for items, which is usually unavailable in practical situations. Our work focuses on providing scalable recommendations for various practical applications in the context of big data. We build a general recommender system that implements a series of algorithms to satisfy different requirements of applications, dealing with the diverse raw user action data, i.e., users' implicit feedback data.

[1] http://storm.incubator.apache.org

3. ARCHITECTURE OVERVIEW

In this section, we first give our solution from the system aspect, including the platform selection, data access solution, and status data storage.

3.1 Platform Selection

In Tencent Inc., the total number of daily active users exceeds 100 million. There are roughly 4 billion user actions and more than 1 PB data generated to be handled every day. In addition, TencentRec needs to deal with 10 billion user requests per day, leading to nearly trillion computations. To support this massive computations, a cluster consisting of thousands of machines is needed, where the machine failures are common. Therefore, the underlying platform to support TencentRec must be scalable and fault-tolerant. Specifically, this claims three requirements. First, it should be able to do real-time computations to capture users' real-time interests. Second, it need to be linearly scalable, easily extended to more machines to support numerous computations. Third, it must be fault-tolerant to serve the practical computations in real-world.

Given these considerations, we compare several real-time computation systems such as Yahoo S4 [2], Spark Streaming [3], and Storm [4] and choose Storm to support our real-time recommendation computations. Storm is a distributed real-time computation system for processing streams of data. We select Storm to construct the real-time recommendation data processing system mainly because of the following considerations. First of all, Storm supports real-time data stream computation. Second, Storm has good scalability that we can dynamically add or delete nodes in the cluster. Third, Storm is fault-tolerant that it is a stateless system which can do fast failure recovery. In addition, Storm has other advantages such as simple programming model, a wide range of programming languages support, and an active open source community that can provide good support for Storm usage. These characteristics of Storm make it a promising choice for TencentRec.

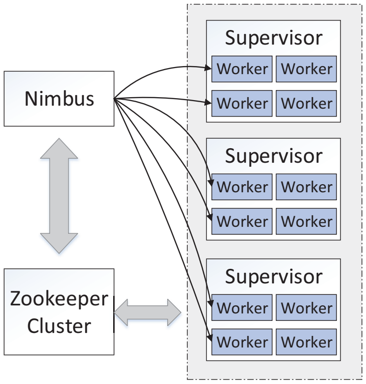


Figure 1: Structure of Storm Cluster

Figure 1 shows the structure of a Storm cluster, which contains two kinds of nodes, i.e., “Nimbus" and “Supervisor". The Nimbus is responsible for distributing code around the cluster, assigning workers to Supervisors, which is similar to Hadoop's [5] “JobTracker". The Supervisor manages worker processes assigned to its machine, starting or stopping worker processes as necessary. Many worker processes spreading across many machines run in parallel to accomplish a computation task. A zookeeper cluster is utilized to coordinate between Nimbus and Supervisors. The Nimbus and Supervisors are designed to be fail-fast and stateless that all states are kept in the zookeeper cluster. In this way, if the Nimbus or Supervisors die, they can restart like nothing happened. Furthermore, no worker process is affected by the death of Nimbus or Supervisors. With this design, the Storm cluster achieves considerable stability and fault tolerance.

3.2 Data Access

TencentRec is designed to be a general recommender system that can serve a wide range of production applications with various characteristics. A key factor is how to access the massive data from different applications in different data formats. We develop Tencent Data Access (TDAccess) to provide a unified interface for various data collection and distribution, decoupling the data sources and the data processing systems. TDAccess gathers data from different applications, and the data processing systems can consume data from TDAccess in real-time.

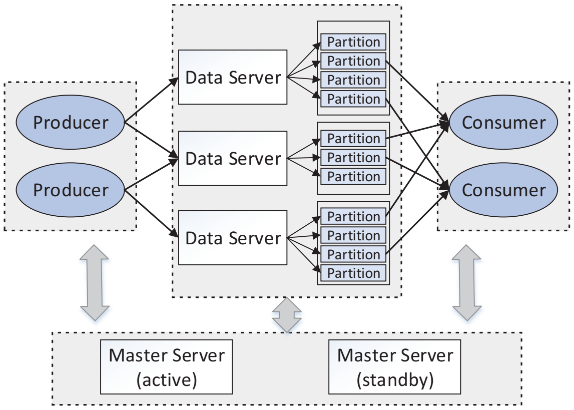


Figure 2: The Architecture of TDAccess

Figure 2 shows the architecture of TDAccess. We apply the Publish/Subscribe model as the communication model. The producers represent the applications, and the consumers are the data processing systems. The data servers are responsible for data cache and the data's publish and subscribe, communicating with the producers and consumers. There are two master servers, an active server and a standby server, to monitor the cluster, keeping the balance between data servers and producers or consumers. The key factors in designing and implementing TDAccess are as follows. First of all, in the TDAccess cluster, data servers do not need to share data that their status are managed by the master server. As a result, the cluster is easily to be linearly scaled for the large amount of data collection and distribution. Second, an important role of the data servers is the message queue between producers and consumers. However, it is not as same as the traditional message queues that usually do not store the message data. To achieve the unconditional availability, TDAccess caches the data in disk in cases such as the temporary absence of the real-time computation systems, or the offline computation requiring the historical data. This will no doubt increase the difficulty of real-time data reads and writes. We utilize sequential operations to accelerate the speed of reads and writes to the largest extent. Third, to achieve better parallelism, we divide the data from an application into many partitions among the data server cluster, as shown in the figure. When the producer or consumer cluster wants to produce or consume data, it first communicates with the master server. The master server will balance the data servers and the producers in the granularity of partition, and return the data server lists to access. Afterwards, the producer or

[2] http://incubator.apache.org/s4/
[3] http://spark.apache.org/streaming/
[4] http://storm.incubator.apache.org/
[5] http://hadoop.apache.org

consumer cluster can communicate with these data servers directly to send or get data in parallelism of partitions.

3.3 Status Data Storage

Storm provides good fault-tolerance as it is a stateless system that can do fast failure recovery. However, the recommendation algorithms like the collaborative filtering and content based methods need to keep status data that record the historical behaviors. The straightforward solution is to store the status data in the workers' memory together with the program. However, it will cause the following drawbacks. On one hand, it limits the design of the program. It is common that some status data are needed by multiple workers, e.g., updated or accessed by different workers. Therefore, we need to manage the distribution and transformation of the status data in programming. Furthermore, we need to maintain the status data's consistency in the program. All these factors make it difficult to design and implement the program. On the other hand, the straightforward solution may degrade the robustness of the system. The failure of workers reserving the status data will cause the program to produce wrong results, since these workers cannot recover their reserved status data. To solve this problem, we utilize a distributed memory-based key-value storage called Tencent Data Store (TDStore) to store the status data used in recommendation computations. In this way, the robustness support of TencentRec is shared by Storm and TDStore. Storm guarantees the running of programs and TDStore is responsible for the status data recovery.

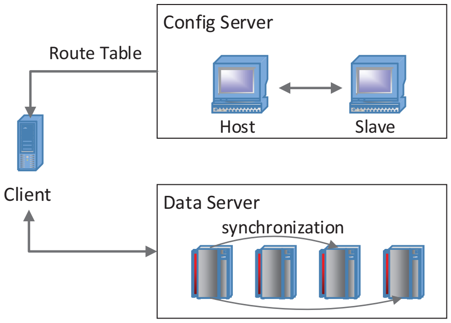


Figure 3: The Framework of TDStore

The framework of TDStore is shown in Figure 3, which is composed of two components, i.e., the config servers and data servers. The config servers consist of a host config server and a backup config server, which manage the route table and track the data servers. The data servers are responsible for data storage, supporting the Memory DataBase (MDB), Level DataBase (LDB)[6], Redis DataBase (RDB) [7], and File DataBase (FDB)[8] storage engines. Each data instance has multiple backups to avoid its loss. The key consideration of TDStore is how to support the large amount of reads and writes. To satisfy this requirement, we do the backup in the granularity of data instance that a data server may be the host server of some data instances but the backup server of others. As a result, almost all the data servers are providing service simultaneously, though only the host data server provides service for a certain data instance. The fine-grained backup promotes the resource utilization. When a client calls a TDStore API, it will first query the host config server to get the route table. After that, it communicates directly with the data servers located by the route table to read or store data. The synchronization between the host data server and the slave data server is done by the host data server and slave data server themselves without the config server's much involvement. After a data instance is updated, the host data server will notify the slave data server, and the slave data server will update its data when idle. In this way, the burden of the config server is lightened.

4. ALGORITHM DESIGN

In this section, we illustrate how we design the recommendation algorithms used in TencentRec. The “real-time", “accurate" and “big" challenges pose multiple requirements for the recommendation algorithms. The “accurate" requires TencentRec to improve the quality of recommendations as much as possible; however, the “real-time" and “big" call for algorithms that are easily to train and scalable for the distributed system. These two requirements tend to be conflicted to some extent. To obtain fast recommendations for the numerous amount of data, the recommendation algorithm should be simple to use, but this tends to reduce the overall quality of recommendations. An efficient recommender system must take both aspects into consideration.

To achieve scalable recommendations, a commonly used method in the real-time recommender system is data sampling [25], which can produce recommendations within limited computational resources but tends to reduce the accuracy of recommendations. It is highly desirable that all available data are used to produce accurate recommendations [16]. Given this consideration, we turn to classic algorithms that can scale for the distributed recommender system and retain their effectiveness, e.g., content based algorithm [18], collaborative filtering algorithm (CF) [16], association rule based algorithm (AR) [24] and situational CTR algorithm (CTR).

We implement the aforementioned algorithms in TencentRec to serve various requirements from production applications. Specially, we employ some mechanisms to make these algorithms more applicable to the production usage, including applying demographic methods to solve the data sparsity problem and some real-time filtering techniques to capture the users' real-time interests. Furthermore, we develop a practical scalable item-based CF method that can be easily implemented in the distributed streaming computation systems like TencentRec. Due to the space constraint, we will just introduce our practical item-based CF algorithm here and take it as example to illustrate the mechanisms employed for production usage.

4.1 A Practical Item-based Collaborative Filtering

Collaborative Filtering (CF) is a popular recommendation algorithm used in industrial systems, including the user-based CF methods and the item-based CF methods. CF tries to recommend items to a user based on his rating behaviors in the past. User-based CF methods generate recommendations based on a few customers who are most similar to the user while the item-based CF methods provide recommendation list for the user by combining the similar items of the items previously preferred by the user. The empirical evidence has shown that item-based CF method can provide better performance than the user-based CF method [8, 26]. We develop a practical item-based CF [16] in TencentRec, and we will describe it in detail below.

To use the item-based CF in practice, we design the algorithm based on the three requirements, i.e., “big", “real-time" and “accurate". Specifically, we consider the following three aspects. First, to achieve the “accurate" goal, we work out the implicit feedback problem. Second, we propose a scalable incremental update mechanism for the real-time item-based CF. Third, we propose a real-time pruning technique to reduce the computation cost.

[6] https://code.google.com/p/leveldb/
[7] http://redis.io/
[8] http://filedb.codeplex.com/

4.1.1 Basic Item-based CF

We first review the basic item-based CF algorithm in this section. Item-based CF mainly contains two main components: item similarity computation and prediction computation.

Item similarity computation is to build a similar-items table by computing the similarity scores for each pair of items. Assume we have n users and m items in the recommender system, the ratings of users for items can be expressed as an $n * m$ matrix R, where each $r_{p,q}$ represents the user u_p's rating for item i_q. An item can be represented as a vector of ratings that n users give to it, i.e., $i_q =< r_{1,q}, r_{2,q}, ..., r_{n,q} >$. There exist many approaches to measure the similarity between items. Here we use cosine similarity as our measure due to its popularity and simplicity. It is defined as follows.

$$sim(i_p, i_q) = \frac{\overrightarrow{i_p} \cdot \overrightarrow{i_q}}{\parallel \overrightarrow{i_p} \parallel \parallel \overrightarrow{i_q} \parallel} = \frac{\sum_{u \in U} r_{u,p} r_{u,q}}{\sqrt{\sum r_{u,p}^2}\sqrt{\sum r_{u,q}^2}} \quad (1)$$

where U denotes the set of users. Specially, if user u does not rate the item i_p, the rating $r_{u,p}$ is considered as zero.

To recommend items for a user, we need to predict the ratings he would give for the unseen items. The prediction of the target item for a user is given by the weighted average rating of its neighbors given by the user:

$$\hat{r}_{u,p} = \frac{\sum_{i_q \in N^k(i_p)} sim(i_p, i_q) r_{u,i_q}}{\sum_{i_q \in N^k(i_p)} sim(i_p, i_q)} \quad (2)$$

where $N^k(i_p)$ is the set of k neighbors, i.e., the k items most similar to i_p.

4.1.2 Implicit Feedback Problem Solution

The key factor of an accurate algorithm is to capture users' preferences for items precisely, which are usually reflected by the user ratings. However, the explicit feedback, e.g., the star ratings for items, is not always available in the practical situations. The majority of the abundant data is implicit feedback, i.e., user behaviors that indirectly reflect user opinions. Since we can only guess users' preferences according to the user behaviors, implicit feedback may reduce the recommender systems' performance if not appropriately handled [12]. In this section, we resolve this implicit feedback problem to provide more accurate recommendations.

There are various types of user behaviors in our scenario, including click, browse, purchase, share, comment, etc. Different user actions represent different degrees of user's interests, and should have different influences on the recommendation algorithm. Therefore, we set different weights to different action types. For example, a browse behavior may correspond to a one star rating while a purchase behavior corresponds to a three star rating. Specifically, for an item that the user demonstrates multiple interests, i.e., multiple behaviors, we take the weight of action with max weight as the user's rating for the item, which can reduce the noise brought by the various messy implicit feedback. The co-rating that a user gives to item i_p and item i_q is defined as the min weight of user-item rating, i.e.,

$$\text{co-rating}(i_p, i_q) = \min(r_{u,p}, r_{u,q}) \quad (3)$$

where $r_{u,p}$ is the max weight of the actions user u takes to the item i_p.

Since the co-rating(i_p, i_q) changes from $r_{u,p}r_{u,q}$ to $\min(r_{u,p}, r_{u,q})$, we need to redefine the similarity between item i_p and item i_q to make sure the similarity score fall in range [0,1], as follows:

$$sim(i_p, i_q) = \frac{\sum_{u \in U} \min(r_{u,p}, r_{u,q})}{\sqrt{\sum r_{u,p}}\sqrt{\sum r_{u,q}}} \quad (4)$$

where the $\parallel \overrightarrow{i_p} \parallel$ in Equation 1 is set to be $\sqrt{\sum r_{u,p}}$ instead of the standard L2-norm.

In addition, for the clarity of description, we say a user rates or likes an item when the user demonstrates implicit interests on the item.

4.1.3 Scalable Incremental Update

Except for the implicit feedback problem, the real-time item-based CF algorithm faces challenge of frequent update of the similar-items table, which is caused by the changing of user ratings. In the traditional recommender systems, the static periodical update strategy is adopted. However, the periodical update cannot meet the requirements of the real-time recommendation, which is a fast-changing scenario. To capture the users' real-time interests, the incremental update strategy is preferred in our item-based CF algorithm.

In [30], the authors proposed an incremental item-based CF method for continuous explicit ratings. Inspired by that incremental update mechanism, we split the similarity score of an item pair into three parts in our practical CF algorithm, as follows:

$$sim(i_p, i_q) = \frac{\text{pairCount}(i_p, i_q)}{\sqrt{\text{itemCount}(i_p)}\sqrt{\text{itemCount}(i_q)}} \quad (5)$$

where

$$\text{itemCount}(i_p) = \sum r_{u,p} \quad (6)$$

$$\text{pairCount}(i_p, i_q) = \sum_{u \in U} \text{co-rating}(i_p, i_q) \quad (7)$$

In this case, the similarity score of an item pair can be computed by pairCount(i_p, i_q), itemCount(i_p) and itemCount(i_q) that all can be naturally incrementally updated. We can independently calculate the new values of pairCount(i_p, i_q), itemCount(i_p) and itemCount(i_q), and combine these values to obtain the value of the new similarity score. As a result, the similarity score can be updated incrementally as follows:

$$\begin{aligned} sim(i_p, i_q)' &= \frac{\text{pairCount}(i_p, i_q)'}{\sqrt{\text{itemCount}(i_p)'}\sqrt{\text{itemCount}(i_q)'}} \\ &= \frac{\text{pairCount}(i_p, i_q) + \Delta\text{co-rating}(i_p, i_q)}{\sqrt{\text{itemCount}(i_p) + \Delta r_{u_p}}\sqrt{\text{itemCount}(i_q) + \Delta r_{u_q}}} \end{aligned} \quad (8)$$

where $sim(i_p, i_q)'$ represents the new similarity score after a new observation (user, item, rating) is received.

We employ three layers to compute the real-time incremental item-to-item similarities in parallel, as shown in Figure 4. First, user action tuples that include the users' ids, items' ids and other information about the actions are grouped by user ids, which enables us to refer to different users' behavior histories in parallel. According to a user's behavior history, we can calculate the new rating given by the user for the item and co-ratings for related item pairs. Besides, the old ratings and co-ratings are saved in the user's behavior history. We can identify these changed ratings or co-ratings that need to be recalculated by comparing the new ratings or co-ratings with the old ones. After that, we transfer the item or item pairs generated together with their changed weights, i.e., Δr_{u_p} and Δco-rating(i_p, i_q) to the next layer. The user rating weights are grouped by the item ids or item pair keys. According to Equation 8, we can update the $itemCount$s and $pairCount$s incrementally, which can be calculated in parallel for each item or item pair. Third, after we get the $itemCount$s and the $pairCount$s, we can calculate the similarity between items using Equation 5. Note that by the key grouping, only a single worker node should operate over a specific item pair at some point. Therefore, the calculation can be safely scaled.

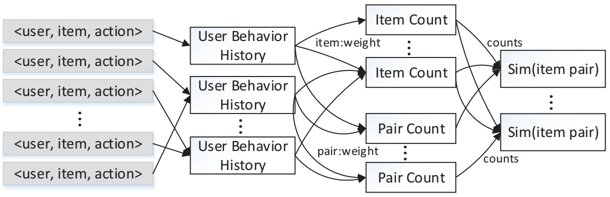


Figure 4: The Multi-layer Item-based CF

4.1.4 Real-time Pruning

The large volume of data pose another challenge on the implementation of real-time item-based CF algorithm. In the item-based CF, two items are considered as related if users tend to rate them together. We set a linked time for the item pairs that two items are considered as related only if users rate them together within a certain period. In TencentRec, there are more than four billion user actions generated everyday. Take news as example, each user has more than ten news rated in average everyday. When the linked time between news is set to six hours, i.e., we generate item pair for two news if they are rated by a user within six hours, there will be ten item pairs generated to update for each user action. Each item pair update needs several computations, resulting in massive computations for big number of user actions. For recommendations in most situations such as e-commerce websites, the linked time is usually set to be three days or seven days, with nearly one hundred item pairs generated for each user action. In that case, each user action leads to hundreds of computations, most of which are not necessary since a large portion of the generated item pairs are not so similar that only the items in $N^k(i_p)$ in Equation 2 are useful for our prediction, thus leads to much resource cost.

To solve this problem, we utilize the Hoeffding bound theory and develop a real-time pruning technique. The Hoeffding bound is used in [10] to build very fast decision trees for data stream analysis. It can be expressed as follows: let x be a real-valued random variable whose range is R (for the similarity score the range is one). Suppose we have made n independent observations of this variable, and computed their mean $\hat{x}$. The Hoeffding bound states that, with probability $1-\delta$, the true mean of the variable is at most $\hat{x}+\epsilon$, where

$$\epsilon = \sqrt{\frac{R^2 ln(1/\delta)}{2n}} \tag{9}$$

In TencentRec, we take the similarity scores of an item pair at different time points as random variables. Let t be the threshold of an item's similar-items list, i.e., the minimum similarity score among the similar items $N^k(i_p)$. Given a desired δ, the Hoeffding bound guarantees that the correct similarity score of the item pair is no more than t with probability $1-\delta$, if n updates have been seen at the moment and $\epsilon < t - \hat{x}$. In other words, after we observe some item pairs' similarity update operations, we can prune the dissimilar item pairs, i.e., the items not able to be in $N^k(i_p)$. Since the similarity computation involves two items, the pruning is bidirectional. We take the min threshold of the two items' similar-items lists as t to do pruning computation. Note that t denotes the minimum similarity score for an item pair (i_p, i_q) to be recorded, i.e., i_q is possible to be in $N^k(i_p)$ only if $sim(i_p, i_q)$ is larger than t. The computation of the item-based CF algorithm with real-time pruning is shown in Algorithm 1, where n_{ij} records the update count of item pair i, j, and L_i denotes the set of pruning items for item i.

Algorithm 1: Item-based CF Algorithm with Real-time Pruning

```
Input: user rating action recording user $u$ and item $i$
Get $L_i$
for each item $j$ rated by user $u$ do
    if $j$ in $L_i$ then
        Continue
    end
    Update pairCount$(i, j)$
    Get itemCount$(i)$ and itemCount$(j)$
    Compute $sim(i, j)$ using Equation 5
    Increment $n_{ij}$
    Get threshold $t_1$ of $i$'s similar-items list
    Get threshold $t_2$ of $j$'s similar-items list
    $t = min(t_1, t_2)$
    Compute $\epsilon$ using Equation 9
    if $\epsilon < t - sim(i, j)$ then
        Add $j$ to $L_i$
        Add $i$ to $L_j$
    end
end
```

4.2 Data Sparsity Solution

The numerous data pose additional challenges for the recommendation algorithms to provide high-quality recommendations. In most recommender systems, the number of ratings already obtained is usually very small compared with the number of user-item pairs. We call this "data sparsity" problem. The data sparsity problem tends to affect the performance of recommender systems, leading to poor recommendations [1]. In big data era, it is more serious than ever. As the users' time is limited, the number of known ratings is finite, but the items are growing in an unprecedented speed, leading to a large user-item matrix with a small number of ratings. As a result, the user-item matrix is more sparse than ever.

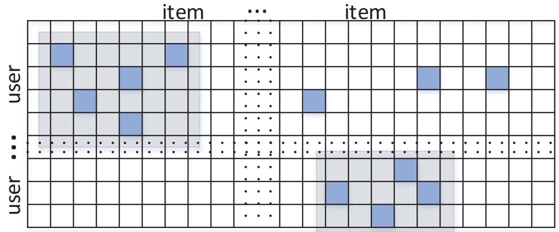


Figure 5: User-item Matrix in Big Data Era

We develop two mechanisms to solve the data sparsity problem, including the demographic clustering and the demographic based complement. First, we cluster users into different demographic groups according to their properties such as gender, age and education. Users in a demographic group generally share similar interests or preferences. As shown in Figure 5, the user-item matrix of a demographic group is obviously less sparse than the global user-item matrix. To run the recommendation algorithms in the demographic user groups, we will get a more refined model and produce more accurate results.

Based on clustered demographic groups, we can rely on the group information to further understand users. Specifically, for users who have few historical data, we propose the demographic based algorithm (DB) [19] to further complement the recommendation results. DB aims to provide recommendations to a user based on his demographic group. To be specific, we compute each demographic group's hot items, i.e., the most popular items in the group. For the situation where the commonly used algorithms like item-based CF and CB cannot effectively generate good recommendations, such as for a new user or those users especially inactive, we can utilize the results of DB algorithm and recommend the hot items to the users.

4.3 Real-time Filtering Mechanisms

Recommendations continuously evolve over time in real-time recommender systems like TencentRec. We utilize two techniques to capture users' real-time interests, including the sliding window and real-time personalized filtering. We capture the global real-time trends with the sliding window technique and use real-time personalized filtering to serve the individual user's real-time demand. These techniques are simple yet effective in maintaining the system's sensitivity to recent data.

Sliding window is a common technique used in real-time data streams computation to "forget" older data [29]. A sliding time window processes the data in the current time window and then it discards a small set of expired data and moves ahead with a small step. In TencentRec, we split the time window into several sessions and we just consider the W most recent sessions in the recommendation computation. For example, implementing the sliding window, the $r_{u,p}$s used to compute the $itemCount$s and $pairCount$s in real-time item-based CF will refer to the ratings given by user u in recent W sessions, as follows:

$$sim(i_p, i_q) = \frac{\sum_{w \in W} \text{pairCount}_w(i_p, i_q)}{\sqrt{\sum_{w \in W} \text{itemCount}_w(i_p)}\sqrt{\sum_{w \in W} \text{itemCount}_w(i_q)}} \tag{10}$$

where the $itemCount_w$ and $pairCount_w$ are the corresponding counts in the wth time session and can be updated incrementally, e.g., $itemCount_w(i_p)' = itemCount_w(i_p) + \Delta r_{u_p}$. In this way, we achieve the sliding time window with the time interval of each session as its sliding step. As data of different types have different life cycles, we provide the flexibility to get recommendations over sliding window of different time intervals. Both the time interval of the overall time window and the small time session can be specified by users, so that we can set different time intervals to suit different requirements of various applications.

Besides the sliding window mechanism, we propose a real-time personalized filtering technique to serve the individual users' real-time demands. For each user, we record the recent k items that he is interested in. Based on the perception that a user's interests fade away as time goes on, we believe only the recent k items are effective for the user's recommendation computation. Therefore, when predicting the unknown ratings, we just consider the recent k items of users to do prediction. Take the item-based CF as example, the $N^k(i_p)$ in Equation 2 will be redefined as the set of most recent k items of the user. Furthermore, if the algorithm cannot produce efficient recommendations in this way, e.g., the item pairs' similarity scores are too low in item-based CF computation, we use the real-time DB algorithm results to complement. Practices show that this real-time complement is more effective than the recommendations based on the old behaviors.

5. IMPLEMENTATION DETAILS

In this section, we will present the implementation details of TencentRec. We first present the overview of our program, and then illustrate how we solve the problems encountered in implementation. Some optimizations in the implementation are also revealed.

5.1 Topology Framework

The primary concept in Storm is stream, which represents an unbounded sequence of data tuples. "Spouts" and "bolts" are the basic operations provided by Storm to conduct stream processing. A spout is responsible for producing the input stream for a Storm cluster, and passing the data to bolts. A bolt may consume any number of input streams and transform those streams in some way. Bolts can also emit new streams and pass them to some other bolts. Complex stream processing, like the item-based CF recommendation algorithm, may require a graph of multiple spouts or bolts where the edges indicate how the streams should be passed around. The graph is called a "topology", which is what we submit to Storm for real-time computation. A topology will process messages forever unless it is killed.

As a general recommender system, TencentRec is designed to provide recommendations for different production applications with various characteristics. These applications pose different requirements to the system, from algorithms to filtering techniques. For example, in news recommendation, the new items keep appearing, and the life span of items is short, in which case the item-based CF algorithm is not so suitable but the CB method applicable instead. In advertisement recommendation, whether a user clicks the advertisement or not is depending on many factors from the advertisement's picture to its placement position, where the CTR (Click Through Rate) predicting algorithms are more efficient than the common recommendation algorithms. Furthermore, applications usually have their own filtering techniques, e.g., the recommended items should be of one specific category or of price within a certain range. To serve all these applications, we need a topology framework that contains all these required algorithms and can be easily scale to serve different applications' filtering requirements, as shown in Figure 6.

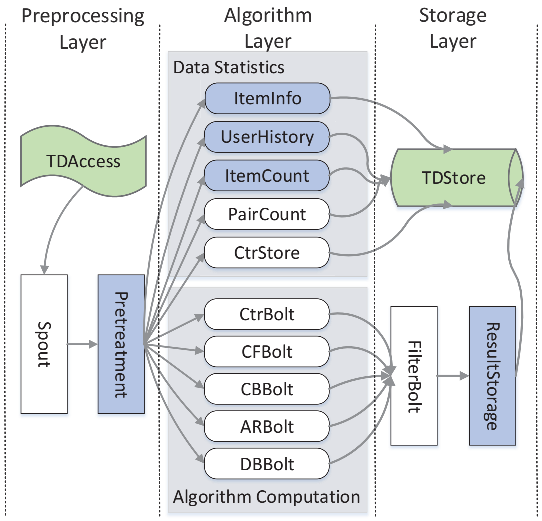


Figure 6: The Topology Framework of TencentRec

The processing of TencentRec can be divided into three major layers.

- The first preprocessing layer gets data from TDAccess, parses the raw message, filters the unqualified data tuples, and transforms data tuples to the next layer.
- The second layer, i.e., the algorithm layer is the main part of TencentRec, which is responsible for the main algorithm computation. We have implemented a number of well-known recommendation algorithms mentioned in Section 4, including the item-based CF, AR, CB, DB, and situational CTR algorithms.
- The storage layer filters the produced results generated by the algorithm layer according to different rules of different ap-

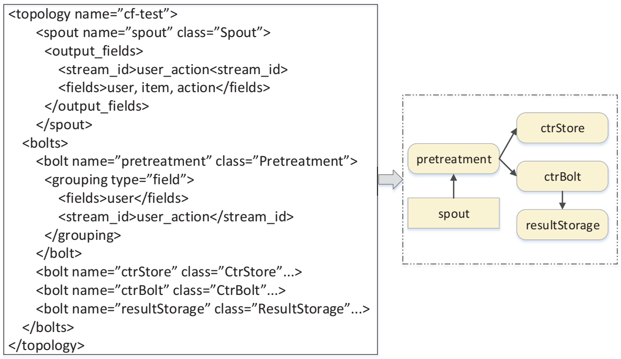
plications, and updates the computation results in TDStore, which can be easily accessed with APIs when answering recommendation queries from applications.

Specially, we divide the computation units (spouts and bolts) used in TencentRec into four types.

- `Application Common Units` refer to the common processing steps of different applications, such as the $Pretreatment$ and the $ResultStorage$. They are represented as the blue grey rectangles in the figure.
- `Application Specific Units` are the rectangle white units, representing the units that are unique for specific applications, such as the $Spout$ and $FilterBolt$.
- `Algorithm Common Units` are the units that are needed by more than one algorithms, usually the data statistics, such as the $ItemCount$, represented by the blue grey rounded rectangles.
- `Algorithm Specific Units` are the white rounded rectangle units in the figure, referring to the units that are specific for algorithms, usually the specific computations, such as the $CFBolt$ and $ARBolt$.

Note that some functions in the topology may be divided into more than one steps, accomplished by several bolts, such as the $CFBolt$. For the sake of clarity of the figure, we combine these steps into one bolt to represent the function.

This distinction between the common bolts and the specific bolts let multiple applications share the common steps and multiple algorithms share the statistical data. Therefore, we can reduce the computation and storage cost as much as possible, thus make the best usage of the resources. Furthermore, as we can see in Figure 6, we decouple the data statistics and algorithm computation in the algorithm layer. Data statistics part computes the statistical results of data tuples and updates them into the TDStore. Algorithm computation part reads statistical data from TDStore and executes the algorithm computation. This design leads to better scalability that we can easily extend or make adjustments to the recommendation algorithms for new application requirements. In addition, it strengthens the robustness of the system. Since all computation units in the topology are state-free, the topology can conduct fast failure recovery with the good fault-tolerance provided by Storm.



```
<topology name=”cf-test”>
      <spout name=”spout” class=”Spout”>
        <output_fields>
           <stream_id>user_action<stream_id>
           <fields>user, item, action</fields>
        </output_fields>
      </spout>
   <bolts>
      <bolt name=”pretreatment” class=”Pretreatment”>
        <grouping type=”field”>
           <fields>user</fields>
           <stream_id>user_action</stream_id>
        </grouping>
      </bolt>
      <bolt name=”ctrStore” class=”CtrStore”...>
      <bolt name=”ctrBolt” class=”CtrBolt”...>
      <bolt name=”resultStorage” class=”ResultStorage”...>
   </bolts>
</topology>
```

Figure 7: An Example XML File and Storm Topology

With the multi-layer structure of topology framework, TencentRec can be capable of providing recommendations for different applications. For each application, we can construct a topology containing the necessary units and its special requirements. All applications run in parallel, not interacting with each other. To deploy different topologies easily, we implement a module to generate Storm topologies from XML configuration files. The XML configuration file states which spouts and bolts it needs and the ways to compose them to construct topology. To generate topology for a specific application, we just need to rewrite the XML file. An example XML file and the generated Storm topology is shown in Figure 7, which implements the situational CTR algorithm, containing one spout and four bolts.

5.2 Temporal Burst Events

The first challenge in implementing TencentRec is the temporal burst. Users have different activities in different time periods of the day. The number of user activities in peak hours can be several times or even hundreds of times more than that of the non-peak period. Consider a situation where a hot news bursts and many users read the news, resulting in a burst of user feedback. It requires large amount of resources to handle the burst of data tuples, dealing with a large number of reads and writes, and large amount of computation. In this case, the large number of reads and writes on the storage units will reduce the TDStore's performance.

We employ the cache technique to solve this problem based on the intuition that user activities in the temporal burst events always have the locality that the small portion of the items attract the large portion of users' attention. We do the fine-grained cache in the granularity of data instance, i.e., a key-value pair, to make best utilization of the cache. The main challenge in implementing the cache is how to maintain data consistency. First, we use “stream grouping" to send tuples with the same key to the same worker to guarantee the validity of the cache. In addition, for workers who update data read by other workers, they first read the data from the cache and then update it both in cache and in TDStore. In this way, we save the read times by the updating worker, and meanwhile, do not affect other workers to use the updated data.

5.3 Hot Item Problem

The next challenge is what we called “hot item problem", referring to visiting inequality problem between hot items and cold ones. Take the news as example, the read time of a hot news is much more than the common news. There will be large number of records of the hot news generated for the computation, e.g., the $itemCount$ statistics. All of these records will be sent over the network to a single worker to calculate or update the news's statistical data. The massive amount of transformation and computation will cause the worker responsible for processing the hot news to be overburdened, leading to the extreme allocation unbalance among the workers or the crash of the worker.

We employ *combiner* technique to solve this problem. The combiner is a map that buffers the coming tuples. When a bolt receives a tuple, it first put it into the combiner, which will do partial merging of the tuples with same key. The combine operation may be increment, addition or maximization. We will fetch the tuples from the combiner and do the costly calculation like TDStore writes at the predefined intervals. Take the $itemCount$ statistics as example, we will observe numerous reading actions that users read the same hot news. We first add these users' rating changed weights in combiner, and then put the overall group rating changed weights into TDStore.

With the combiner, the number of reads and writes can be reduced significantly. Since the combine operations are much lighter than the reads and writes, the system's performance will be largely improved. In addition, in a temporal burst situation, the combiner's efficacy will be even improved. This is because that the combiner combines the data tuples with same key in specified time intervals.

As a result, the use ratio of the combiner map will increase in the temporal burst events.

5.4 Other Optimizations

We employ some additional techniques in implementing TencentRec to accelerate the real-time computation and reduce the resource usage. One of the most important optimizations is the multi-hash technique. It is proposed to solve the write conflict problem in implementing TencentRec. The problem arises when we do the demographic user group count statistics, e.g., $itemCounts$ and $pairCounts$. Actions of users in one group may not be distributed to the same bolt, since their user ids are different. In this case, each bolt will send an $itemCount$ or $pairCount$ update request to the TDStore, resulting in multiple write requests from different workers, i.e., the write confliction. A plain solution is to use lock mechanism in the TDStore writing. However, this will complicate the programming and reduce the performance of the TDStore. We employ a multi-hash strategy to solve this problem. For each user action tuple, we first hash it to the corresponding bolt according to the user id. After updating users' ratings for items, we hash the rating changed weights by user group ids to next bolt that ratings of the same user group will go to the same bolt. The next bolt receives the user rating changed weight data tuples and updates the group's rating for the item.

6. DEPLOYMENT AND PERFORMANCE

With the rich algorithms implemented, TencentRec can be easily used in a wide range of production applications, such as e-commerce websites, news, videos, mobile applications, and advertisement recommendations in social networks, etc. In Tencent Inc., TencentRec has been applied to handle different types of recommendation tasks, as shown in Figure 8.

Figure 8: Some Applications Applying TencentRec

6.1 Deployment

In this section, we introduce how TencentRec is deployed to serve various applications, and briefly illustrate how recommendations are produced.

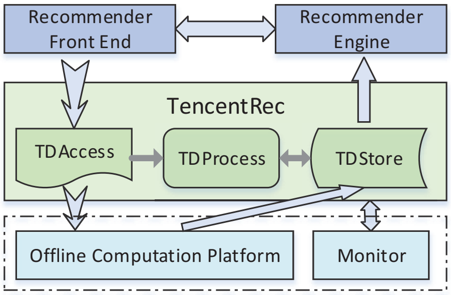


Figure 9: The Deployment of TencentRec

Figure 9 shows the deployment of the TencentRec in production applications. Specially, we use "TDProcess" to represent the data processing component based on Storm in the figure, which can help express the usage of TDAccess and TDStore more clearly. The TDProcess gets data streams from various applications with the help of TDAccess. The computational results of TDProcess are saved in TDStore. The recommender front end is responsible for interacting with users, i.e., dealing with the user queries, and displaying the recommendations to users. The recommender engine accepts user queries preprocessed by the front end and utilizes the computing results in TDStore to generate the recommendation results. The front end accepts the recommendation results and displays recommendations to users in proper form. In addition, we can add other components in the deployment easily, such as the offline computation platform to do offline computations and the monitor to get an overview of the system running.

There are no special requirements for the computers used to deploy the TencentRec clusters. Some computers are equipped with one four-core processor and 32 GB memory, and other configurations include computers with two four-core processors and 8 GB memory, two six-core processors and 64 GB memory, and two six-core processors and 128 GB memory. In our deployment, some applications share one common cluster and some applications are each served by one single cluster. The biggest cluster contains nearly 200 computers, and the overall size of the clusters is about 1500 computers. The overall clusters deal with 4 billion user action tuples per day, with data size of more than 1 PB. There are 10 billion user requests every day, with maximum 0.5 million requests in one second. Each request requires 50 computations in average, leading to nearly trillion computations for TencentRec. Nevertheless, it can provide real-time recommendations steadily.

6.2 Performance Evaluation

In this section, we take several representative applications as example to evaluate the efficiency of TencentRec, including the Tencent News [9], Tencent Videos [10], YiXun [11](an e-commerce website supported by Tencent), and advertisement recommendation in QQ [12](One of the most popular social networking services in China developed by Tencent).

To evaluate the performance of TencentRec, each application provides recommendations to some users by their own original methods and the others using the new TencentRec recommendation approach, and records their performance separately. The original recommendations are provided by offline computation or the semi-real-time computation, without the real-time filtering mechanisms. Algorithms used in our system vary for different applications, as shown in Table 1. The choice of algorithms is done manually according to different applications' characteristics, since the recommendation in real world is complex and is affected by many factors. If there are more than one possible methods which are hard to select, an A-B test will be taken for a certain period to decide which algorithm to use. For example, we use CB to do news recommendation because of the rich content information and the emerging new items. For advertisement recommendation in QQ, the situational CTR algorithm is utilized to predict the CTR in different situations. In addition, we employ item-based CF to produce recommendations in video websites and e-commerce websites. Specially, since the DB algorithm is used by all applications to do complement, we eliminate it in the table. To make the performance comparable for different applications, we take the CTR (Click Through Rate) of recommendations as the performance metric to evaluate, such as the CTR of goods in YiXun, the CTR of news in Tencent News, and the CTR of videos in Tencent Videos.

[9] http://news.qq.com/
[10] http://v.qq.com/
[11] http://www.yixun.com/
[12] http://im.qq.com/

Table 1: Overall Performance Improvement

	Algorithms	Performance Improvement (%)		
Applications		**avg**	**min**	**max**
News	CB	6.62	3.22	14.5
Videos	CF	18.17	7.27	30.52
YiXun	CF	9.23	2.53	16.21
QQ	CTR	10.01	1.75	25.4

We collect the data of these applications in one month and show the overall performance of TencentRec in Table 1. Due to the privacy concern, we just show the overall performance improvement of TencentRec but eliminate the specific numbers, i.e., the improved percentage of the recommendation CTR. From the table we can see that TencentRec has improved the recommendation CTRs significantly. Specially, it performs excellently in videos recommendation, with an 18.17% improvement in average and a maximum value of 30.52%.

6.3 Performance in News Recommendation

To further illustrate the superiority of TencentRec, we choose two applications to show their performance in one week in detail. In this section, we first demonstrate the efficiency of TencentRec applied in Tencent News.

Tencent News is a news website provided by Tencent Inc. for users to get latest news. We collect its data in one week and compute two indicators to show the performance of TencentRec. The website recommends users with latest news, and users will click or read them if they are interested in these news. The two indicators we compute are the CTR of recommended news and the average read count per user, both of which reflect the effectiveness of recommendations. The original recommendations the application provides are generated by the semi-real-time computation, that the CB recommendation model is updated once an hour. As a result, it cannot provide real-time recommendation for users to get the latent tendency.

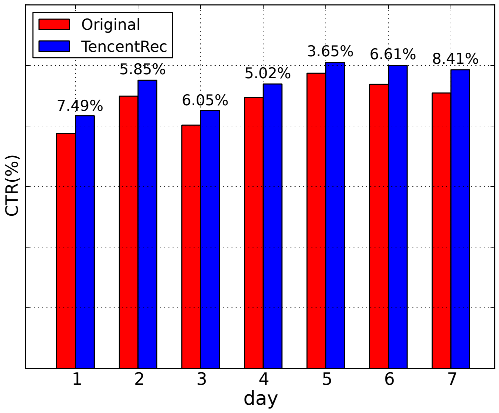


Figure 10: CTR of Tencent News in One Week

Figure 10 shows the performance of TencentRec in TencentNews with the CTR as its metric. The numbers in the figure represent the day CTR improvements brought by TencentRec. The performance taking read count per user as metric is shown in Figure 11. We eliminate the specific numbers of CTR or read count in the figures due to the privacy concerns. From the figures, we can see that TencentRec performs better than the original one steadily for both of the CTR and read count per user, with maximum CTR improvement of 8.41%.

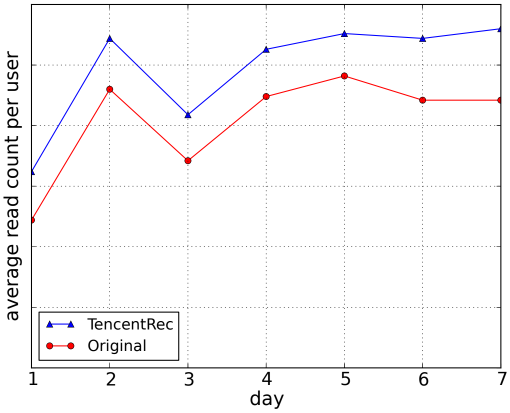
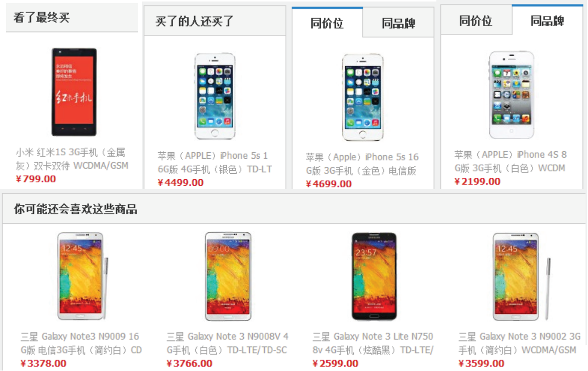


Figure 11: Average Read Count per User of Tencent News in One Week

6.4 Performance in E-commerce Recommendation

The other example application we choose to evaluate is YiXun, an e-commerce website established in 2006. YiXun provides various kinds of commodities including the electronic products, daily necessities, etc. As shown in Figure 12, there are a number of recommendation positions in YiXun which provide recommendations with different features, e.g., the goods with similar prices, the goods with similar tags, the goods with similar purchases. We collect data of YiXun in one week, and show the performance in two positions, i.e., similar price recommendations and similar purchase recommendations. When a user browses one commodity, the similar price recommendations display the commodities with similar price that user may like, while the similar purchase recommendations produce recommendations based on other users' similar purchase behaviors, provide other commodities that are purchased by the users who have also purchased this commodity.



看了最终买

小米 红米1S 3G手机（金属灰）双卡双待 WCDMA/GSM
¥799.00

买了的人还买了

苹果（APPLE）iPhone 5s 16G版 4G手机（银色）TD-LT
¥4499.00

同价位 同品牌

苹果（Apple）iPhone 5s 16G版 3G手机（金色）电信版
¥4699.00

同价位 同品牌

苹果（APPLE）iPhone 4S 8G版 3G手机（白色）WCDM
¥2199.00

你可能还会喜欢这些商品

三星 Galaxy Note3 N9009 16G版 电信3G手机（简约白）CD
¥3378.00

三星 Galaxy Note 3 N9008V 4G手机（白色）TD-LTE/TD-SC
¥3766.00

三星 Galaxy Note 3 Lite N7508v 4G手机（炫酷黑）TD-LTE/
¥2599.00

三星 Galaxy Note 3 N9002 3G手机（简约白）WCDMA/GSM
¥3599.00

Figure 12: Different Recommendation Positions in YiXun

We take the CTR of recommendations as indicator to evaluate the performance of different methods, as shown in Figure 13 and Figure 14. The original methods in the figures generate the recommendations offline with a lot of filter conditions, e.g., price, browse count and purchase count, etc., and the model is updated once a day. In TencentRec, utilizing the computation ability, we compute the real-time statistical data in different dimensions and granularities. As a result, we can provide more refined recommendations. After obtaining the candidate commodities generated by item-based CF, we first check the user's real-time demands that whether the user is recently interested in some candidates. If interested candidates are not found, we rank the candidates based on the DB algorithm, providing recommendations based on how his similar users rate these commodities. For the user who does not have the information like gender or age, we will use the global demographic group to produce recommendations.

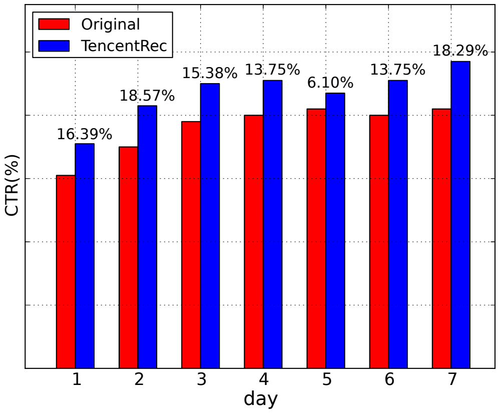


Figure 13: CTR of Similar Price Recommendation in YiXun

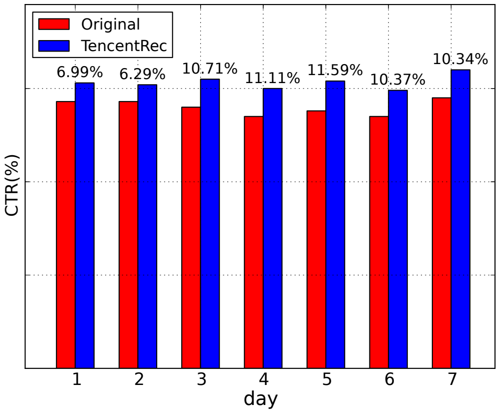


Figure 14: CTR of Similar Purchase Recommendation in YiXun

From the figures, we can see the superiority of TencentRec in YiXun's recommendations, which performs better than the original method steadily in both advertisement positions. Specially, we find that TencentRec gains a higher improvement in the similar price recommendation than the similar purchase recommendation. This is because that the similar purchase recommendation provides users with items based on users' purchase history, where we have relatively explicit preferences about the user. However, in the similar price recommendation, the data we rely on to generate recommendations are sparse that the candidate items are very large, and we should utilize the massive implicit feedback to produce recommendations. As a result, the TencentRec that can capture users' real-time interests based on the practical algorithm implementation and data sparsity solution will get significantly better performance. This is important in the production usage since a user's opinion about an application is largely dependent on his first time experience where he has few information for the application to use.

7. CONCLUSION AND FUTURE WORK

In this paper, we have presented a comprehensive recommender system called TencentRec to provide general real-time recommendations for a wide range of production applications. We exploited the features of Storm to deal with the real-time stream computation, and developed TDAcess and TDStore to handle the various data access and massive status data storage respectively. We implemented several classic algorithms to produce accurate real-time recommendations, including the well-known recommendation algorithms commonly used in industry such as the collaborative filtering, content based, and demographic based algorithm. In addition, we employed some mechanisms to make these algorithms more practical and accurate in the real-time recommendations. Specially, we proposed a practical item-based CF algorithm, with the super characteristics such as robust to the implicit feedback problem, scalable incremental update and real-time pruning. TencentRec is implemented in a refined manner, utilizing several optimizations including fine-grained cache, combiner function and multi-hash technique. With a multi-layer topology framework, TencentRec is easily to be deployed in a series of production applications. By now, TencentRec has dealt with about 4 billion real-time user action data tuples and 10 billion recommend requests per day for a long time, with about 1 trillion computations every day. By collecting data from several online business applications, we have demonstrated the superior performance of TencentRec.

Though TencentRec does well in handling the real-time data stream recommendations of diverse applications, there still are several future works we can do. First, the parallelism of the spouts and bolts in Storm topology is set manually at present. It is desirable for TencentRec to set the parallelism automatically according to the data size of specific applications. Second, we plan to provide more machine learning techniques used in recommender systems in later TencentRec.

8. ACKNOWLEDGMENTS

We thank all the Tencent Recommender Platform Team members for their contributions to this paper. Many thanks to Yong Li, Kunqian Hong, Yu Xiang, Zhijun Chen, Zhao Xu, and Chong Du for their kind suggestions and support.

The research is supported by the National Natural Science Foundation of China under Grant No. 61272155 and 61272340, Beijing Natural Science Foundation(4152023), 973 program under No. 2014CB340405 and Tecent Research Grant (PKU).

9. REFERENCES

[1] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Trans. on Knowl. and Data Eng.*, 17(6):734–749, June 2005.

[2] D. Agarwal, B.-C. Chen, and P. Elango. Fast online learning through offline initialization for time-sensitive recommendation. In *Proc of the 16th ACM SIGKDD Conference*, pages 703–712, 2010.

[3] B. Chandramouli, J. J. Levandoski, A. Eldawy, and M. F. Mokbel. Streamrec: A real-time recommender system. In *Proc of the 2011 ACM SIGMOD Conference*, pages 1243–1246, 2011.

[4] C. Chen, H. Yin, J. Yao, and B. Cui. Terec: A temporal recommender system over tweet stream. *Proc. VLDB Endow.*, 6(12):1254–1257, Aug. 2013.

[5] W.-Y. Chen, J.-C. Chu, J. Luan, H. Bai, Y. Wang, and E. Y. Chang. Collaborative filtering for orkut communities:

Discovery of user latent behavior. In *Proc of the 18th WWW Conference*, pages 681–690, 2009.

[6] B. Cui, H. Mei, and B. C. Ooi. Big data: the driver for innovation in databases. *National Science Review*, 1(1):27–30, 2014.

[7] A. S. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: Scalable online collaborative filtering. In *Proc of the 16th WWW Conference*, pages 271–280, 2007.

[8] M. Deshpande and G. Karypis. Item-based top-n recommendation algorithms. *ACM Transactions on Information Systems (TOIS)*, 22(1):143–177, 2004.

[9] E. Diaz-Aviles, L. Drumond, L. Schmidt-Thieme, and W. Nejdl. Real-time top-n recommendation in social streams. In *Proc of the 6th ACM RecSys Conference*, pages 59–66, 2012.

[10] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proc of the 6th ACM SIGKDD Conference*, pages 71–80. ACM, 2000.

[11] W. Hill, L. Stead, M. Rosenstein, and G. Furnas. Recommending and evaluating choices in a virtual community of use. In *Proc of the SIGCHI Conference*, pages 194–201, 1995.

[12] Y. Hu, Y. Koren, and C. Volinsky. Collaborative filtering for implicit feedback datasets. In *Proc of the 2008 8th IEEE ICDM Conference*, pages 263–272, 2008.

[13] X. Jin, Y. Zhou, and B. Mobasher. A maximum entropy web recommendation system: Combining collaborative and content features. In *Proc of the 11th ACM SIGKDD Conference*, pages 612–617, 2005.

[14] Y. Kang and N. Yu. Soft-constraint based online lda for community recommendation. In *Advances in Multimedia Information Processing-PCM 2010*, pages 494–505. Springer, 2011.

[15] B. M. Kim, Q. Li, C. S. Park, S. G. Kim, and J. Y. Kim. A new approach for combining content-based and collaborative filters. *Journal of Intelligent Information Systems*, 27(1):79–91, 2006.

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

[17] T. Miranda, M. Claypool, A. Gokhale, T. Mir, P. Murnikov, D. Netes, and M. Sartin. Combining content-based and collaborative filters in an online newspaper. In *Proc of ACM SIGIR Workshop*, 1999.

[18] R. J. Mooney and L. Roy. Content-based book recommending using learning for text categorization. In *Proc of the 5th ACM DL Conference*, pages 195–204, 2000.

[19] M. J. Pazzani. A framework for collaborative, content-based and demographic filtering. *Artif. Intell. Rev.*, 13(5-6):393–408, Dec. 1999.

[20] A. Popescul, D. M. Pennock, and S. Lawrence. Probabilistic models for unified collaborative and content-based recommendation in sparse-data environments. In *Proc of the 17th UAI Conference*, pages 437–444, 2001.

[21] S. Rendle and L. Schmidt-Thieme. Online-updating regularized kernel matrix factorization models for large-scale recommender systems. In *Proc of the 2008 ACM RecSys Conference*, pages 251–258, 2008.

[22] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl. Grouplens: An open architecture for collaborative filtering of netnews. In *Proc of the 1994 ACM CSCW Conference*, pages 175–186, 1994.

[23] F. Ricci, L. Rokach, and B. Shapira. *Introduction to recommender systems handbook*. Springer, 2011.

[24] J. J. Sandvig, B. Mobasher, and R. Burke. Robustness of collaborative recommendation based on association rule mining. In *Proc of the 2007 ACM RecSys Conference*, pages 105–112, 2007.

[25] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Analysis of recommendation algorithms for e-commerce. In *Proc of the 2nd ACM EC Conference*, pages 158–167. ACM, 2000.

[26] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Item-based collaborative filtering recommendation algorithms. In *Proc of the 10th WWW Conference*, pages 285–295, 2001.

[27] A. I. Schein, A. Popescul, L. H. Ungar, and D. M. Pennock. Methods and metrics for cold-start recommendations. In *Proc of the 25th ACM SIGIR Conference*, pages 253–260, 2002.

[28] U. Shardanand and P. Maes. Social information filtering: Algorithms for automating &ldquo;word of mouth&rdquo;. In *Proc of the SIGCHI Conference*, pages 210–217, 1995.

[29] J. Vinagre and A. M. Jorge. Forgetting mechanisms for incremental collaborative filtering. In *Proc of 2010 WTI Workshop*, 2010.

[30] X. Yang, Z. Zhang, and K. Wang. Scalable collaborative filtering using incremental update and local link prediction. In *Proc of the 21st ACM CIKM Conference*, pages 2371–2374. ACM, 2012.

[31] M. Ye, X. Liu, and W.-C. Lee. Exploring social influence for recommendation: A generative model approach. In *Proc of the 35th ACM SIGIR Conference*, pages 671–680, 2012.

[32] H. Yin, B. Cui, L. Chen, Z. Hu, and Z. Huang. A temporal context-aware model for user behavior modeling in social media systems. In *Proc of the 2014 ACM SIGMOD Conference*, pages 1543–1554, 2014.

[33] H. Yin, B. Cui, Y. Sun, Z. Hu, and L. Chen. Lcars: A spatial item recommender system. *ACM Trans. Inf. Syst.*, 32(3):11:1–11:37, July 2014.

[34] H. Yin, Y. Sun, B. Cui, Z. Hu, and L. Chen. Lcars: A location-content-aware recommender system. In *Proc of the 19th ACM SIGKDD Conference*, pages 221–229, 2013.