Technische Universität Berlin

Master Thesis

Serving Large Machine Learning Models on Apache Flink

Submitted to the Faculty IV, Electrical Engineering and Computer Science Database Systems and Information Management Group in partial fulfillment of the requirements for the degree of Master of Science in Computer Science as part of the ERASMUS MUNDUS programme IT4BI

Ziyad Muhammed Mohiyudheen

Matriculation #: 387646

Supervisor: Prof. Dr. Volker Markl
Advisor: Behrouz Derakhshan

15/08/2017

Erklärung (Declaration of Academic Honesty)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

I hereby declare to have written this thesis on my own and without forbidden help of others, using only the listed resources.

15/08/2017

Datum

Ziyad Muhammed Mohiyudheen

Contents

List of Figures

List of Tables

1. English Abstract

With advancement in technologies for large scale data processing, it is now possible to train large machine learning models at affordable cost and time. This has led to more feature rich models in areas such as personalized recommendations and targeted online advertisement. However most of the research in this area has been focused on the training of large models. Research community has now started to address a critical component in large scale machine learning pipeline – model serving and management. Model serving enables real-time predictions for different end-user applications and near real-time updates to the model for the newly observed data. In this thesis, we propose a novel model serving approach for large machine learning models using the real-time stream processing and queryable state features offered by Apache Flink. We show how to deploy large matrix factorization and classification models trained in any platform and how online updates and periodic retraining can be performed on the deployed model. The solution provides scalable and fault tolerant model serving with low latency predictions for different end-user applications.

2. Deutscher Abstract

Mit dem Fortschritt in Technologien für die Verarbeitung von großen Datenmengen (”Big Data”) ist es nun möglich, umfangreiche Modelle des Maschinellen Lernens mit erschwinglichen Kosten- und Zeitaufwand zu trainieren. Dies hat zu mehr Modellen mit vielfältigen Eigenschaften in Bereichen wie personalisierter Empfehlungen und gezielter Online-Werbung geführt. Indes konzentriert sich das Gros der Forschung in diesem Bereich auf das Training umfangreicher Modelle. Die Forschungsgemeinschaft hat nun begonnen, eine kritische Komponente in Data-Pipelines zu betrachten - Modellserving und -management. Modellserving ermöglicht Echtzeitvorhersagen für verschiedene Endbenutzeranwendungen und Nah-Echtzeitaktualisierung des Modells für die neu beobachteten Daten. In dieser Arbeit postulieren wir eine neue Herangehensweise des Modelservings für umfangreiche Modelle des Maschinellen Lernens mittels der Real-Time-Stream-Processing- und Queryable-State-Features die Apache Flink bietet. Wir zeigen auf, wie große Matrixfaktorisierungs- und Klassifikationsmodelle von verschiedenen Trainingsplattformen eingesetzt und Online-Updates und periodisches Neutraining auf dem eingesetzen Modell angewandt werden können. Diese Anwendungsweise liefert skalierbares und fehlertolerantes Modelserving mit kurzfristigen Vorhersagen für verschiedene Anwendungen für Endbenutzer.

3. Introduction

Machine learning [11] techniques are employed in various applications in the digital world today. It powers complex data products such as personalized recommendations in online shops, weather forecasting, contextual advertisement on the web and detecting credit card fraud. On a high level, a machine learning task involves two activities – model training and model serving. Model training applies a learning algorithm on the available data to learn a model. Model serving uses the learned model for making predictions on the new data.

Model training tasks include cleaning the data and perform necessary pre-processing such as feature selection and normalization. The data is split into training and test sets. The learning algorithm is applied on the training set to learn the model. The produced model is then evaluated for performance on the test set. These steps are repeated by tuning the input parameters of the learning algorithm until we obtain satisfactory performance for the model.

The produced model is useless unless it can be used in different end-user applications for making predictions. Also, the model needs to be continuously evaluated for quality based on the new observations, triggering a retraining if the quality is not satisfactory. Model serving handles this deployment and management of the trained model.

The research community has focused heavily on the model training, resulting in various machine learning algorithms suitable for different types of data products. Advancement in the research of large scale data processing (known as big data systems or platforms) has enabled the training of machine learning models on very large data sets instead of using a sample of the data. Both of these have led to feature rich models and hence high performance data products.

However, the research on model serving is relatively new. Most of the enterprises still depend on custom solutions for model serving [21]. The trained model is often moved to a low latency database so that end-user applications can query this model. Incremental updates to the model with the new data (online learning) is done partially or often not performed, instead a retraining of the entire model is carried out in scheduled frequencies [31]. Moreover, this approach demands huge effort in deploying multiple models.

Research community is now trying to address these issues with general purpose model serving frameworks. The goals for such frameworks are: deploying multiple machine learning models with minimum effort, serving end-user applications with low latency and high throughput predictions and performing incremental updates to the model. The model serving frameworks should be scalable and fault tolerant as well.

This thesis introduces a model serving framework specifically designed for serving large machine learning models, satisfying all of the above mentioned goals. We present a novel approach

based on stateful real-time stream processing and queryable state features offered by Apache Flink [27] and reliable message queues offered by Apache Kafka [36]. Our modular design makes it extensible to other technologies which can offer similar features.

We evaluate the solution looking at mainly two quality attributes – latency and throughput. Latency refers to the time required for serving a single prediction request from an external client. Throughput measures the number of prediction requests served in unit time. We also measure the time required to deploy models of different sizes and improvement in quality of the model in online learning. We show how our solution is able to deploy the models quickly, perform online updates to the model and provide low latency and high throughput predictions to end-user applications.

In the remaining part of this section, we present: (i) a motivating example, (ii) the main challenges for serving large models based on the motivating example, (iii) the main contributions of our work, and (iv) the outline for the remaining sections of the thesis.

3.1. A motivating example

Consider the example of an online book catalogue store, with a large collection of books and a large number of registered users. The catalogue store suggests books to its users based on the explicit feedbacks in the form of ratings and implicit feedbacks in the form of clicks or reading history. The Figure 1 shows the recommendations shown to a user who is looking for science fiction books. The list of books shown to the user is based on the books he had already added in the past and on other readers' choices who have similar interests as that of the user. This suggestion service is an example of a recommender system.

To build this recommender system, we select one of the most popular approaches - collaborative filtering using the matrix factorization [35]. We start with a training data set, which contains the available ratings of different books by the registered users. Here we see the problem as a set of users and set of books and a very sparse rating matrix that contains all the known user-book ratings.

The model learns d-dimensional vectors $w_u \in \mathbb{R}^d$ for each user u and $x_i \in \mathbb{R}^d$ for each book i, where d is the total number of latent factors. The latent factors represent the unknown properties of the users and books. The user vectors are collected to a matrix $W \in \mathbb{R}^{|users| \times d}$ and the book vectors are collected to a matrix $X \in \mathbb{R}^{|songs| \times d}$. The latent factors are learned by solving the below optimization problem.

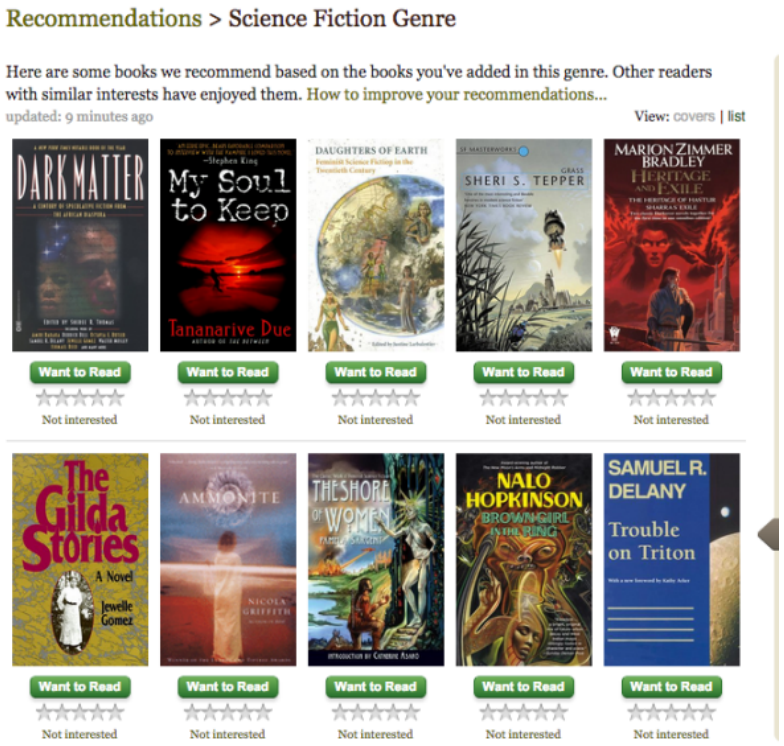

Figure 1: Recommendations in an online book catalogue store[1]

$$\underset{W,X}{\operatorname{argmin}} \lambda(||W||_2{}^2 + ||X||_2{}^2) + \sum_{(u,i)\in Obs} (r_{ui} - {w_u}^T x_i)^2 \tag{1}$$

The unknown rating for any user-book pair can be approximated by taking the dot product of the corresponding user and book vectors. i.e. for any user u, and book i, the predicted rating is given by,

$$rating(u,i) = {w_u}^T x_i \tag{2}$$

The size of this model is proportional to the number of users, number of books and the number of latent factors used in the learning. To serve such a model for predictions in different end-user applications, there are multiple challenges. We discuss these challenges in the below section, and try to identify the main features required in the model serving layer.

[1] Source: goodreads.com

3.2. Main Challenges

If the book store wants to see the predicted rating for any user-book pair, the simplest strategy is to pre-compute all the predictions and materialize the results in a low-latency database. This is not efficient for large number of users and books, as only a small portion of this materialized predictions might be actually used. An efficient approach would be to provide **low-latency access to the model parameters** (in this case, the user and book vectors) by the model serving layer, so that the application can calculate the required user-book rating on the fly. A related requirement is that model serving layer should also be able to serve multiple clients at the same time providing **high throughput**. In case of a failure, we should be able to bring the model serving layer back in action within no-time, making **robustness** another important feature.

As the new rating data arrives, the model should be updated in real-time so that most up-to-date recommendations can be provided to the users. This feature is known as **online learning**, and is becoming one of the sought after features for a model serving layer. One of the key challenges in recommender systems is how to **address cold start**, where we get new users or new books for which no rating data is available yet. Model serving layer should be able to incorporate new users and books efficiently and in real time.

An ideal serving layer should be **platform agnostic**. In other words, the model serving layer should be able to deploy the model trained on any platform of user's choice. After this initial deployment, it is important to check the model quality from time to time. A complete retraining of the entire model should be performed if the quality falls below a defined threshold. Thus model serving layer should also support **model quality evaluation** and **periodic retraining**.

Often the book suggestion service considers other model outputs along with the collaborative filtering model. For example, a classification model to determine the profile of the users can further improve the recommendations. Model serving layer should be able to **serve multiple models**, all **at large scale**. What makes such a serving layer very appealing is, users should be able to deploy new models with minimum effort and still providing all the above mentioned features.

Based on the above discussion, a brief summary of the important features for model serving layer is given below:

- Provide low latency access to the model
- Provide high throughput serving
- Robustness
- Provide online updates to the model

- Provide means to evaluate the model quality continuously
- Be agnostic to the model training platform
- Support periodic retraining and reloading of the model

3.3. Contributions

The contributions of this thesis go as follows:

1. This thesis proposes a novel approach for serving large machine learning models using real time stateful streaming in Apache Flink. Though our solution is implemented in Flink, the same should be extendible to any real-time streaming platform with stateful stream processing capabilities.
2. We show the robustness, low latency and high throughput model serving using matrix factorization recommender model, and how online updates and periodic retraining can be performed on the model using our solution.
3. We show how to extend our solution to other machine learning models by using model serving implementation for a large support vector machine model.

3.4. Thesis Outline

Section 4. Background and related work: In Section 4, we explain the main concepts of model serving in the context of large machine learning models. We provide a survey through the related publications and discuss the advantages and disadvantages of each of them.

Section 5. Design and Implementation: In Section 5, we discuss our solution in detail. The section is divided into: (i) Overview of the technologies used and selected machine learning models, (ii) System architecture, where we describe the details of our solution design, (iii) Implementation, where we describe the specifics of our implementation for the selected machine learning models.

Section 6. Evaluation: In Section 6, we present the evaluation of our approach. First, we introduce the datasets used, the experiment setup and the quality metrics for evaluation. Then we present the experiments conducted and provide detailed analysis of the results.

Section 7 and 8. Conclusion and future work: in the last two sections, we present the conclusions of our work and recommendations for future work.

4. Background and related work

In this section we describe in detail the concepts related to serving large machine learning models and provide necessary background information. We provide a survey of the related work on large scale model serving and explain the advantages and disadvantages of each approach.

4.1. Machine learning

Machine learning is the field of computer science which gives computers the ability to learn and predict without being explicitly programmed [11]. Machine learning techniques are employed in various applications in today's digital world, which includes recommendations in online shops, weather forecasting, filtering spam emails, serving ads on the web and detecting credit card fraud. With advancement in technologies for large scale data processing (known as big data systems or platforms), it is now possible to train the machine learning models on very large data sets, at affordable cost and time. This has led to more feature rich models which perform better.

Machine learning tasks can be classified into the below broad categories [24].

1. **Supervised Learning**

   In supervised learning, a training dataset with known labels are used. Here the goal is to learn a general rule that maps the input training data to the output labels.

2. **Unsupervised Learning**

   No labeled examples are used in unsupervised learning. The learning algorithm tries by its own to find the patterns in the input data. Unsupervised learning could be used to identify the hidden patterns or as an initial step to identify the features before carrying out supervised learning.

3. **Semi supervised learning**

   Semi supervised learning algorithms combines both labeled and unlabeled data in order to learn the general rule to describe the data.

4. **Reinforcement Learning**

   In reinforcement learning, the machine learning algorithm learns how to act based on a given observation. Every action has an impact on the environment and the environment provides feedback to guide the learning algorithm.

5. **Transduction**

   Transduction does not construct a function explicitly, instead it tries to predict new outputs based on training inputs, training outputs and fresh input data.

6. **Learning to learn**

   This category of algorithms tries to learn its own inductive biases based on the previous experience.

The output of a machine learning algorithm is called machine learning model (or simply, model). We can classify the learning tasks based on the nature of the produced model, as given below.

1. **Classification**

   In this type of machine learning, the learning algorithm produces a model which can classify the input data to one of the two (binary classification) or more (multi-label classification) predefined classes.

2. **Regression**

   In regression, the output is a continuous value rather than a set of predefined discrete classes. Both classification and regression are supervised learning tasks.

3. **Clustering**

   Clustering algorithms divide the input data to a set of groups or clusters which are not known beforehand. The elements in a cluster are more similar to each other in some way than to the elements in other clusters. Clustering is an example of unsupervised learning.

4. **Density Estimation**

   This category of machine learning algorithms finds the density distribution of inputs in some n-dimensional space.

5. **Dimensionality Reduction**

   Here the inputs are mapped to a lower-dimensional space, thus reducing the number of features.

The phases in the machine learning pipeline can be broadly divided into the below two categories.

1. **Model Training**

   We discussed about the various types of machine learning algorithms above. Model training phase is the actual execution of these algorithms on the input data to produce the desired model.

2. **Model Serving**

   Once the model is obtained after training, the model should be used for predictions on the new data. As new training data comes in, the model should be updated as well. These functionalities are addressed in the model serving phase.

In this thesis we focus on model serving, specifically for large machine learning models. In the below section, we explain the notions of large machine learning models and model serving in detail.

4.2. Large Machine Learning Models

With the help of big data processing platforms and parallelized version of machine learning algorithms, it is now possible to train on very large datasets than using only a sample from the dataset. However, this large scale machine learning need not necessarily produce a large model. Producing Large models depends on type of the machine learning application, type of the algorithm and the number of features. In this section we discuss two example applications which produces large models.

1. **Recommender Systems**

   Recommender systems are one of the core components in ecommerce and entertainment industries. In both cases recommendations for products or content are provided to the users based on the explicit (ratings) and implicit (click data, purchase history) feedback. Recommendation systems can be broadly classified into three categories based on the approach for building the models – collaborative filtering, content based filtering and hybrid filtering.

In collaborative filtering, the model is created from the user's past behavior such as purchase data, click data and ratings given for different items. The resulting model is then used for predicting the rating for any given user-item pair. The collaborative filtering algorithms uses unknown features, called latent factors. Thus an 'understanding' of the items is not required to build the model. Collaborative filtering works based on the assumption that people who have agreed in the past will agree in the future and people will continue liking the items which are similar to the items they have liked in the past.

In content based filtering approach, known characteristics of items (for example, genre of a book) and a profile of the user (the type of the items the user likes) are used in building the model. Content based algorithms recommend items that have similar known features to those, that a user has liked in the past. Hybrid systems combine both these approaches while building the recommender models.

Collaborative filtering models are an ideal example of large scale recommender model in the settings where large number of users and items are present. We have already discussed this in detail in the Section 3.1, where we described a matrix-factorization approach for collaborative filtering. We also observed that as the number of latent factors increases, the model becomes very large.

2. **Online Advertisement**

   Online advertisement is a multi-billion-dollar industry today, and is one of the most widely used application of machine learning models. Real-time bidding advertising, sponsored search advertising and contextual advertising have all been using different machine learning models for predicting the ad click-through rates (CTR) [41]. A typical industrial model in production may serve different end-user applications and provide predictions on billions of events per day. It is also typical to have large feature space for such models.

   One of the important parameter in predicting CTR is the profile of the user. Various machine learning models are used to classify the user into predefined segments, and ads are shown based on which segments the profile of the user falls into. One example for such a classifier is Support Vector machines (SVM) model, which takes only numeric features. Converting various categorical features to numeric features explode the total number of features making the SVM model really large. SVMs are inherently binary classifiers. In the case of multi-label user profiling, one way to employ SVM is to have a set of binary classifiers and choose the one which classifies the data with the highest margin. This approach is known as one-versus-all classification. Another approach is to build set of one-versus-one classifiers, and chose the class that is selected by most classifiers. In both approaches, the total size of the model becomes very large.

4.3. Model Serving

As the data and features increases, the size of the machine learning model also increases. Often big data frameworks are used only for the training of these large models and then it is moved to the serving layer, which typically resides outside of the big data framework. Incremental updates to the model is often not performed, instead a retraining of the entire model is carried out in scheduled frequencies. This is not ideal for many applications. The research community is now trying to address this issue to have end-to-end machine learning systems within the big data frameworks, that can perform training and updating of the models and serve various external applications.

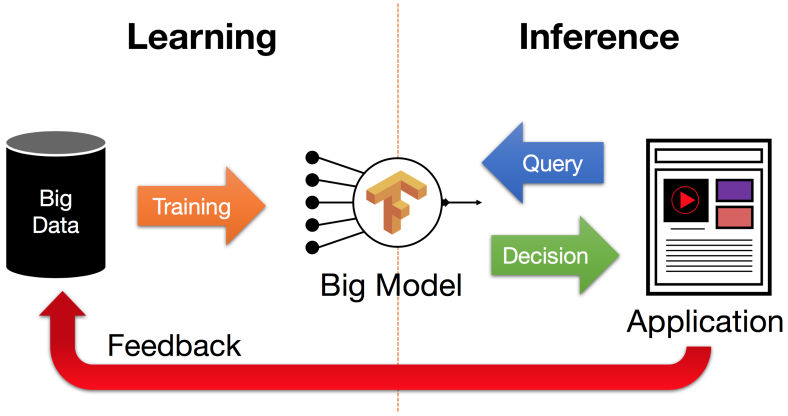

Figure 2: Model Serving of Large machine learning models [14]

The two approaches [34] towards model serving are:

1. **Eager Serving**

   In Eager, we compute and materialize all the predictions and access them in query time. The advantage of Eager serving is that we can use offline training frameworks for efficient batch prediction computation and use traditional data serving systems for serving the applications. The obvious disadvantage is that it demands frequent and heavy computations to reflect model updates, and can serve only when the set of possible queries is limited.

2. **Lazy Serving**

   in Lazy serving, prediction computation happens in real-time. It has the advantage that computation is performed only for the necessary queries which need not be bound to a known set, and rapid updates to the model are possible. The disadvantage is that lazy serving is complex and involves substantial computation overhead of the serving system. It requires low and predictable latency from the model serving layer.

In general, Lazy serving is ideal for large machine learning models, if we can achieve the low latency (less than 10 milliseconds) and high throughput.

4.4. Model Serving Frameworks

In this section we describe some of the pioneer works related to serving of large machine learning models.

4.4.1. Velox

Velox [31] tries to address the deployment of large machine learning models in Berkeley Data Analytics Stack (BDAS) [17]. Velox is a model management system which facilitates online model management, maintenance and serving. Velox proposes an initial offline batch training to produce the model using Apache Spark [46], and the trained model is then deployed using Velox.

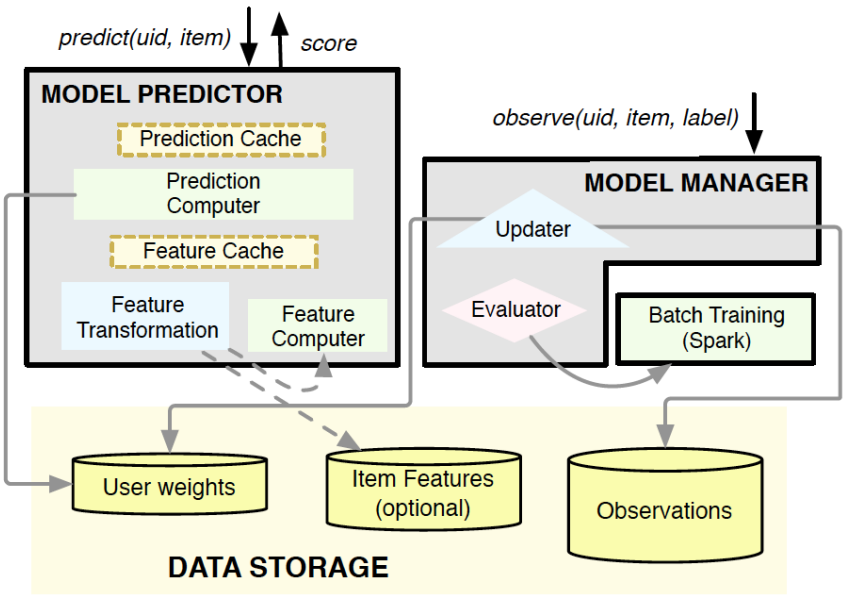

Figure 3: Velox Architecture [31]

The primary components in Velox architecture are:

1. **Velox model manager**: orchestrates the computation and maintenance of pre-declared machine learning models, allows incorporating feedback and new observations, continuously evaluate model performance and trigger the retraining as the quality of the model decreases.

2. **Velox model predictor**: low-latency, up-to-date prediction interface to the deployed model for any application which uses this model. Model predictor also uses some pre-materialization strategies for efficient serving of the model.

In the prototype, Velox supports matrix factorization models which uses latent factor vectors. An example is a song prediction recommender system. The user and song vectors of the model are deployed on Tachyon [38], the default storage system used in Velox. An API is provided to the applications to query these latent factors, in order to obtain the prediction for any user-song pair. As the new observations arrive through the observe API, Velox model manager updates the user vectors, and also check the model quality in order to determine if a retraining is required. Only user vectors are updated in an online fashion, where as the batch retraining updates the item vectors and potentially user vectors as well.

Velox prediction service uses a Feature cache and a Prediction Cache in order to reduce the user and item feature access from the storage layer. The prediction cache is used to cache the final prediction for a given user-item pair where as the feature cache efficiently partition and replicate the materialized feature tables. Velox make use of the fact that while the total number of items might be large, item popularity often follows a Zipfian distribution, resulting in more frequent access of only a few items. So Velox suggests caching only the 'hot' items using a simple cache eviction strategy such as LRU, which will have a high hit rate.

For adding the new users, Velox uses the recent estimate of the average of the existing user weight vectors. Predictions made using the model serving can influence the decisions, which may in turn be used in the retraining of the models. This can lead to feedback loops. In order to overcome these feedback loops, Velox rely on form of a bandit algorithm, which assign an uncertainty score to each item on top of its predicted score.

Though Velox address the deployment of large models, the big data platform in consideration - Apache Spark, is only used in the initial training and batch retraining of the model. Other components of Velox reside outside of Apache Spark. Online updates are carried out only for one component in the model (user vectors). Velox project is already deprecated. The successor of Velox is a more advanced model serving solution, called Clipper [32].

4.4.2. Clipper

Clipper [32] introduces caching, batching and adaptive model selection techniques in a modular architecture. It simplifies the model deployment, with low prediction latency and high throughput, across multiple machine learning frameworks and applications.

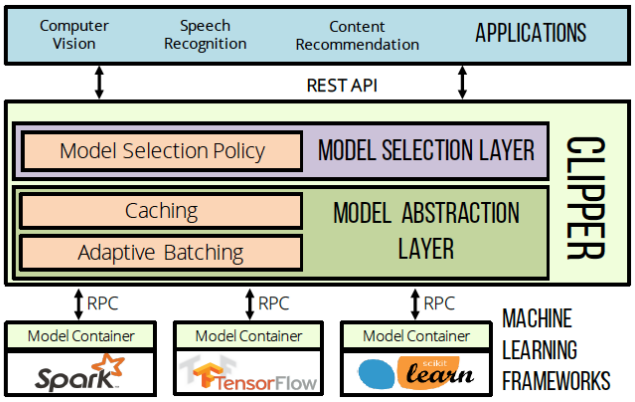

Figure 4: Clipper Architecture [32]

The Clipper architecture consist of two layers:

1. **Model abstraction layer**: This layer provides a common prediction interface and ensures resource isolation. It optimizes the query workload for batch oriented machine learning frameworks underneath. At the top of this layer is the prediction cache, which enables a partial pre-materialization mechanism for frequent queries. The batching component resides below the prediction cache. It assembles the point queries to optimal dynamic batch sizes to be sent to the model containers of specific machine learning framework. The mini-batch of queries to the model containers is submitted via RPC. To achieve process isolation each model is deployed in a separate Docker [23] container.

2. **Model selection layer**: This layer is responsible for selecting one or more of the deployed models and combine their outputs to provide more accurate results. This layer incorporates feedback from the application layer with the predefined model selection policies. It enables selection of the right model(s) for the query submitted through REST/RPC APIs from the application layer.

Clipper way of model serving is suitable if there are multiple machine learning models across different frameworks to be considered for the prediction request, as it provides a common prediction interface for different applications. However, Clipper does not address the challenges in deploying large machine learning models, as well as retraining or updating the model is also

not addressed in this system. Also Clipper does not optimize the execution of models within the machine learning framework where the model is trained and deployed.

4.4.3. LASER

LASER [21] is an end-to-end machine learning solution to deploy response prediction models based on logistic regression, used as part of a social network advertising system. LASER is built on the principle that building machine learning systems should not be only based on the prediction accuracy of the models. Instead it should address a broad spectrum of challenges such as training the models on commodity clustered hardware to reduce the cost, test new models on live traffic data with minimum effort, scalability of run-time computation under latency constraints, continuous evaluation of model quality and quick termination of below par models.

The logistic regression model in LASER is partitioned into a purely feature based cold start component and a warm-start component specific to the ad-campaigns. The cold-start component is trained entirely offline, where as fast retraining is performed on the warm-start component frequently.

LASER ensures the system performance by bringing certain optimizations. In order to reduce the computation in run-time, LASER pre-computes certain interaction terms between user and ad-campaign features. For serving real-time predictions LASER take the 'better wrong than late' approach. i.e. If computing a feature vector is taking time, in order to process the request in the required time window, LASER will substitute the feature vector with a pre-computed mean vector. In such cases a slightly less accuracy is considered preferable to not being able to compute a prediction because data takes too long to arrive. In order to speed-up the dot product calculation between feature vectors, LASER uses a partial result cache, which stores the product of the terms which occur unchanged across multiple feature vectors.

The deployment complexity is addressed using a fixed model choice (logistic regression) and a configuration language based on JSON to specify feature construction and transformation structure. This configuration language restricts feature transformations to be built from the component library and allows for changes in pipeline without service restarts or code modification.

The major drawback of LASER is that it is a highly tailored system to meet the requirements of online advertisement for a specific social network platform. It serves models based only on logistic regression, and the model management is done through a custom configuration language. This tight coupling between the model serving system and model family does not allow LASER to be extended as a general model serving layer.

4.4.4. Oryx2

Oryx2 [13] is a model serving system built using the lambda architecture with Apache Spark and Apache Kafka. It includes end-to-end applications for collaborative filtering, classification, regression and clustering packaged in the distribution. The architecture of Oryx2 is shown in the Figure 5. It consists of three side-by-side cooperating layers of the lambda architecture and a connecting data transport layer.

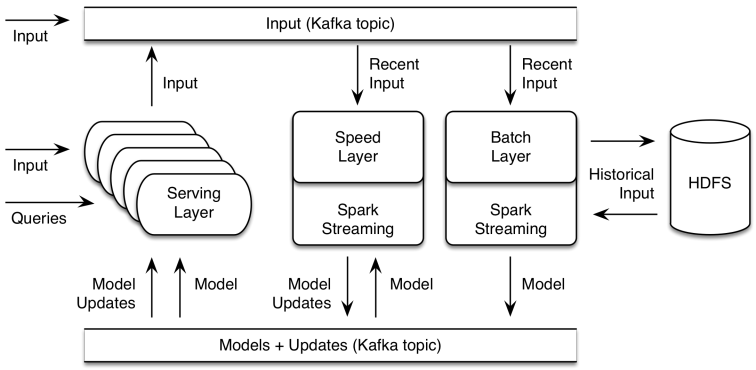

Figure 5: Oryx 2 Architecture [13]

1. **Data Transport Layer**: Data transport is handled using Apache Kafka, primarily as two topics. An 'input topic' to serve the fresh training data and a 'model topic' for storing the model and its updates.

2. **Batch Layer**: It is implemented as a Spark Streaming on Hadoop cluster. The streaming job reads the fresh training data from the input Kafka topic and write to HDFS. The Spark training job reads this fresh data from HDFS along with the historical data already stored on HDFS and use it for training the model. The produced model is written to HDFS as well as Kafka model topic.

3. **Speed Layer**: This is also implemented in Spark streaming, which reads the model from Kafka model topic, keeps the model state in memory and for the fresh data arriving from Kafka input topic, it produces model updates and write back to Kafka model topic.

4. **Serving Layer**: Serving layer listens to Kafka model topic, for any updates to the model. The model is loaded to memory and is used to serve different end-user applications by using an HTTP/REST API on top of the methods that query the model in memory.

Oryx2 is a step forward in model serving, and is in active development. Currently it supports ready-to-deploy machine learning applications for the below use cases:

- Recommendation model based on Alternating Least Squares
- Clustering based on k-means
- Classification and regression based on random decision forests

Though Oryx2 addresses a variety of features for the model serving layer, a disadvantage is that the model in the serving and speed layer is managed in the memory. For serving very large models with billions of features, this is not ideal. Also, Oryx2 does not address the fault tolerance and failure recovery of the model serving explicitly.

4.4.5. PredictionIO

PredictionIO [28] is an open source machine learning sever which allows users to quickly build and deploy predictive engines for any machine learning task using customizable templates. At the time of writing this thesis, PredictionIO is an Apache Incubating project under the Apache license, version 2.0.

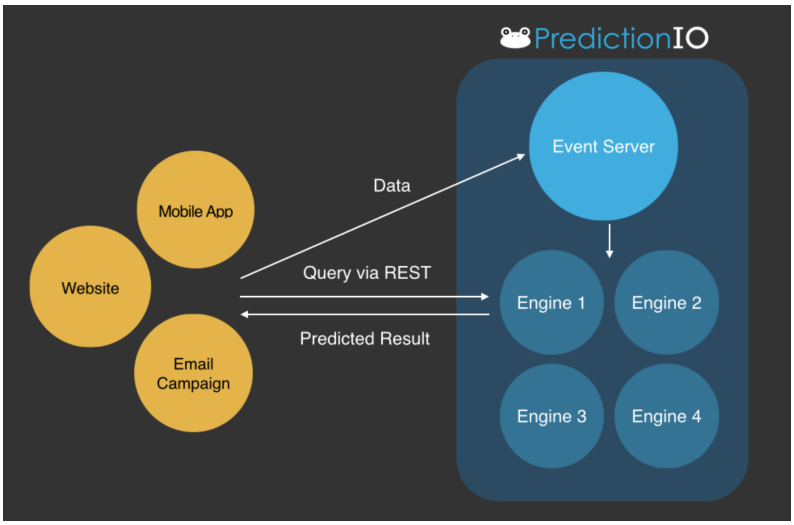

Figure 6: PredictionIO components [19]

PredictionIO consists of below components:

1. **Event Server**: Event server continuously collect data from different end-user applications in real time or in batch. After data collection, event server serves two functions – i) provide data to engines for training and evaluating the models. ii) provide a unified view for data analysis.

2. **Engines**: A predictionIO engine builds predictive models with one or more machine learning algorithms using the data from event server. Then the model is deployed as a web-service. It then listens to queries from applications and provide real-time predictions.

A system architecture of these components is presented in the Figure 7. PredictionIO is primarily built with HBase, Apache Spark, HDFS and Elasticsearch.

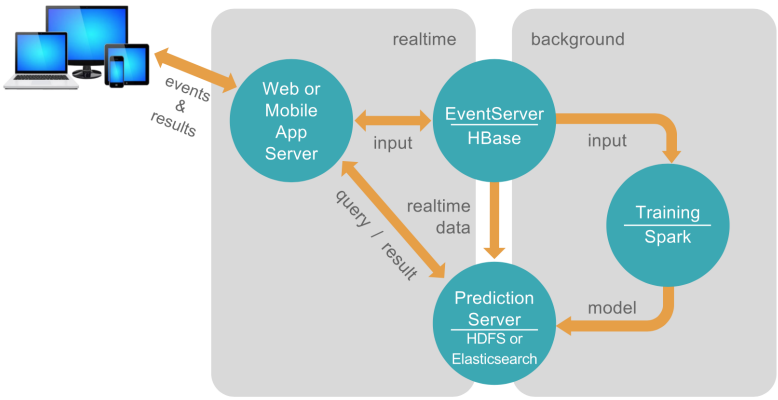

Figure 7: PredictionIO System Architecture [19]

Event Server uses Apache HBase as the data store. Apache Spark with MLlib forms the core of machine learning model training on PredictionIO. The output of training has two parts – model and its metadata. The produced models are either stored in HDFS or Elasticsearch based on the template used in building the machine learning application. The metadata such as model versions and engine versions are stored in Elasticsearch.

PredictionIO provides end-to-end machine learning in a single system, using the state of the art open source technologies. It provides most of the features required for a model serving system. Though online learning is currently not addressed, it is feasible to customize the templates to have online learning capabilities as well. The modular design helps to include additional features where applicable, using the provided templates as a starting point.

4.4.6. TesnorFlow Serving

TensorFlow Serving [9] is a high performance, flexible model serving system designed for production environments. It offers out of the box integration for TensorFlow [20] models. It can also be extended for serving other types of machine learning models. Multiple models including multiple versions of the same model can be served simultaneously. The architecture of TensorFlow Serving supports real time atomic predictions as well as bulk-processing jobs

to pre-compute predictions or analyze the model performance. TensorFlow Serving uses a batch scheduler that groups individual prediction requests from multiple clients. In the presence of hardware accelerators such as GPUs, it further reduces the cost of predictions. The main components in TensorFlow Serving are:

1. **Servables**: This is the central abstraction in TensorFlow Serving. It could be part of a model, a complete model or a tuple of inference models. Clients make the prediction request to the Servables. TesnorFlow Serving allows one or more versions of a Servable to be deployed concurrently. clients are allowed to submit the prediction request to a specific version. This is very useful for A/B testing and gradual rollout.

2. **Loaders**: They are used to manage the life cycle of Servables. Irrespective of the content of a Servable, the Loader API provide a common infrastructure to load and unload Servables.

3. **Source**s: They are the plugin modules that originates Servables. The Source interface allows to use arbitrary storage systems from which Servables can be loaded.

4. **Managers**: They listen to Sources and track all versions of a Servable. Managers provide a simple interface for clients to access the loaded Servables.

This solution is ideal for serving TensorFlow models. For serving other kinds of machine learning models, the users have to implement the above components specific to the machine learning model in consideration.

4.5. Parameter Servers

Parameter Servers [39] are used in large scale machine learning training in order to hold the global shared state of parameters. Though they are studied in the context of large scale machine learning training, their features seem a good fit for model serving as well.

In Parameter Server approach, computational nodes are partitioned to worker nodes and server nodes, as shown in the Figure 8. Each worker node owns a portion of the training data and workload. The server nodes distribute the globally shared parameters (also known as the parameter vector) among them and make it available for the worker nodes. The machine learning algorithm operates iteratively. In each iteration, each worker node independently uses the portion of training data available to it. It fetches the required parameters from server nodes and determine the changes to the parameter vector in order to get closer to an optimal value. Since each worker node's update reflect only its training data, the system needs a mechanism to mix these updates. This is handled using a sub-gradient, a direction in which the parameter vector

should be shifted. System aggregates all sub-gradients before updating the parameters on the server nodes.

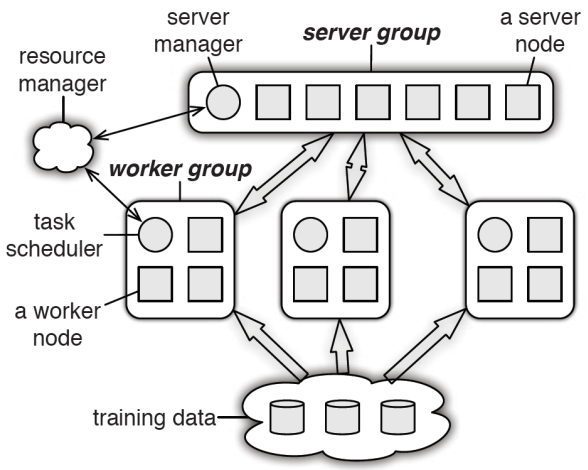

Figure 8: Parameter Server - Architecture [39]

The first generation of parameter servers [44] were using memcached distributed key-value store as a synchronization mechanism. Key-value stores have very large overheads and synchronization cost were expensive and not always necessary. The second generation of parameter servers [22] were application specific which failed to factor out common difficulties between different types of machine learning problems. Also it was difficult to deploy many algorithms in parallel.

The third generation of parameter servers [39] addresses these issues. The shared parameters are presented as (key, value) pairs in order to facilitate linear algebra operations. They are distributed across a group of server nodes. For reliability and scaling, server nodes communicate with each other to replicate and migrate the parameters. The server nodes can push its parameters and pull parameters from remote nodes. User defined functions are supported on server nodes, allowing them to also execute tasks and workloads. The tasks are executed asynchronously and in parallel. The algorithm designer can choose the desired consistency model for parameter updates. Multiple algorithms can be executed in parallel as well.

While the distributed model training is in progress, nodes which host online services consuming this model, can continuously query the model parameters from the server nodes. Another set of worker nodes can simultaneously update this model as the new training data arrives. In this way, parameter server is partially acting as a model serving framework as well. In addition to this, the features of parameter servers such as the elastic scalability, fault tolerance, asynchronous communications and flexible consistency are all relevant for serving large models. We borrow these ideas from parameter server framework in our solution design.

5. Design and Implementation

In this section we present our approach for serving large machine learning models using real-time stateful stream processing. Primarily we use the below two features in the design of our solution:

1. Stateful real-time stream processing
2. Exposing the state for external queries

Both these features are offered by Apache Flink [27], and hence it becomes an ideal choice for our solution. Along with Apache Flink, we use other state of the art open source technologies – HDFS [43] and Apache Kafka [36]. We try to achieve a modular design with pluggable components for our model serving solution, which can be extended to various large scale machine learning models.

In the reminder of this section, first we provide an overview of the technologies used in our design, then we present the architecture of our solution in detail. Finally, we discuss the implementation of our solution on selected machine learning models.

5.1. Overview of Technologies

In this section we provide an overview of the technologies used in our solution, highlighting their features which makes them ideal components in our solution.

5.1.1. Apache Flink

Apache Flink [27] is a unified batch and stream processing framework for big data which offers real time stream processing as well as batch processing on top of its streaming dataflow engine. Flink provides low latency, high throughput streaming engine with exactly-once processing guarantees. In this section, first we discuss the run time environment of Flink and how it executes the streaming jobs. Then we discuss the state management and fault tolerance mechanism in Flink. Finally, we discuss about how Flink exposes its state for external applications using queryable state feature.

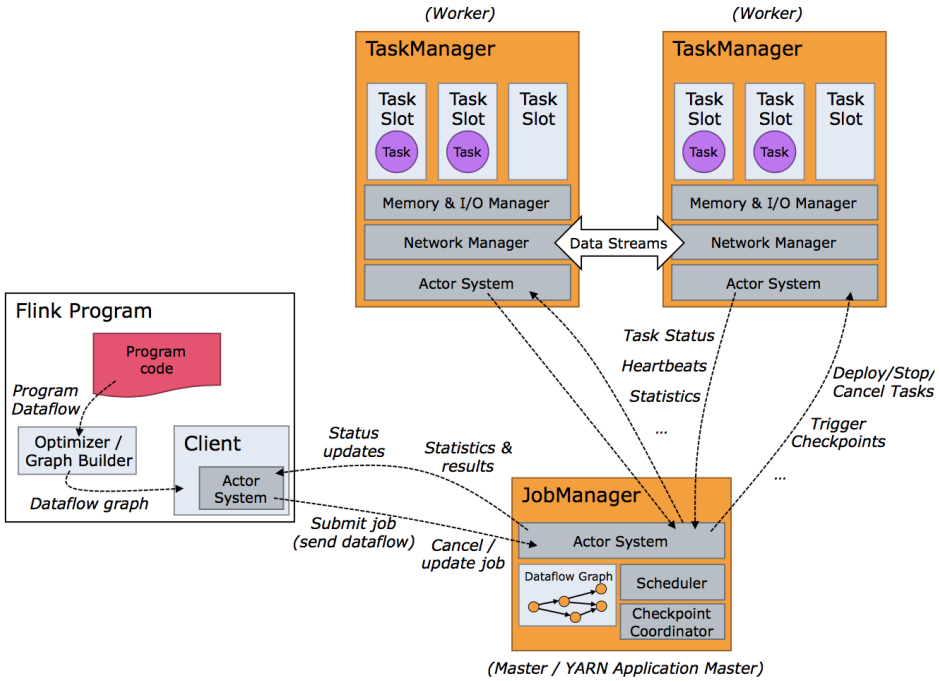

Figure 9: Distributed Run-time environment in Flink [4]

The distributed runtime environment of Flink (shown in Figure 9) consist of 2 major components: JobManager and the TaskManager(s). Flink client program prepares the data flow graph and submit to the JobManager. JobManager coordinate the distributed execution. It is responsible for scheduling the tasks, checkpoint coordination and failure recovery. One or more TaskManagers execute the actual data flow tasks in distributed manner, and work in the master-slave architecture with JobManager.

In High availability mode, more than one JobManagers may be configured, one of them act as the active master, and others are in standby. When a failure occurs to the active JobManager, the standby JobManager takes over the control. If high availability is not configured, JobManager will be the single point of failure. JobManager and TaskManagers can be started directly on the machines, or using a resource management framework such as YARN [45].

Each TaskManager is a JVM process, which can have one or more task slots. The managed memory of the TaskManager is shared between its task slots and is reserved per slot. That means, a subtask running on one of the task slots cannot compete with other subtasks running on remaining slots for managed memory. How ever there is no CPU isolation in this architecture.

5.1.1.1. State and Fault tolerance in Flink

The Figure 10 shows stream processing with and without state. Stateful operators store data across the processing of individual events in a stream. Flink provides stateful functions and operators, thus Flink streaming applications can maintain the value, aggregation or summary of data that has been processed over time.

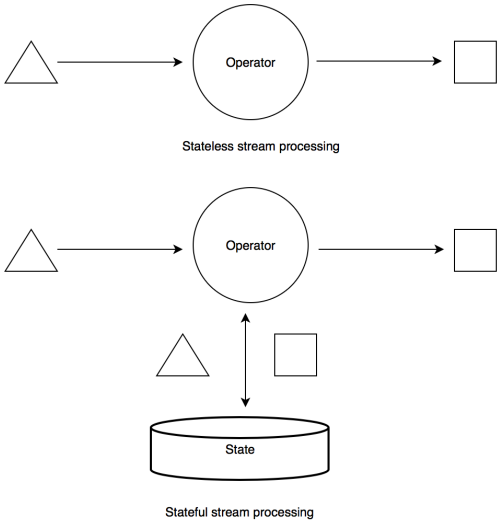

Figure 10: State in Stream Processing

The state in Flink can be classified into keyed state and operator state. On a keyed stream, Flink provides state per key per operator instance. Keys are grouped into key-groups such that at any given time, each key is part of only one key-group and each parallel instance of a keyed operator works with the keys of one or more key-groups. In Operator state, each operator state is bound to one parallel instance of the operator.

Both these states can further be classified into either managed state or raw state. In managed state, data structures controlled by the run-time in Flink are used. In raw state, operators keep their own data structures.

In Flink, the operators are pipelined, and a check pointing mechanism is used to store the result data of each operator in order to be used in case of a failure. The downstream operators in Flink do not have to wait until this checkpoint is materialized. The fault tolerance mechanism used in Flink is called Asynchronous Barrier Snapshotting (ABS) [26] inspired by the standard Chandy-Lamport algorithm [29]. Control records called barriers are injected into the input stream at the stream sources. They flow with the records as part of the data stream. A barrier separates the

records into two categories- the ones belonging to current snapshot and the ones belonging to the next snapshot.

When an intermediate operator receives barriers from all its input stream, it takes a snapshot of the data from the previous barrier to the current one in an asynchronous manner, and it emits a barrier corresponding to the current snapshot to all its output streams. This process is continued in all the operators till the sink operator (end of the data flow). Once a sink operator receives barriers from all input streams it acknowledges that snapshot to the checkpoint coordinator. When all the sink operators acknowledge a snapshot, it is considered complete. Flink depends on this distributed consistent snapshots [27] for its failure recovery. Upon failure, Flink selects the latest completed snapshot, and re-deploy the entire data flow with each operator having the state that was snapshotted as part of the latest completed snapshot.

5.1.1.2. Queryable State

Flink allows managed keyed state to be made queryable for applications which reside outside of Flink. This eliminates the need of having distributed operations or transactions with external systems such as key value stores, thus avoiding certain bottlenecks. To understand the use of queryable state, consider the below example.

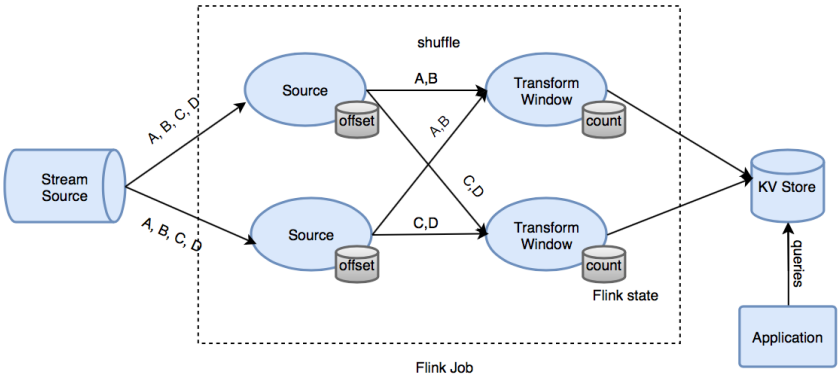

Figure 11: Stream processing with KV Store as sink[2]

Figure 11 shows a Flink stream job, which consumes events (for example, click counts of items) from a stream source, and counts by key (for example, item ID) over a time window and sends the aggregate count after each window to a KV store. The end user application queries the KV

[2]adapted from: https://data-artisans.com/blog/queryable-state-use-case-demo

store for getting the counts per item at any given time. In this job, the aggregate counts are actually present in the state of the window operator. If there is a mechanism to query this state, we could avoid sending the data to the KV store which is a bottleneck in many cases.

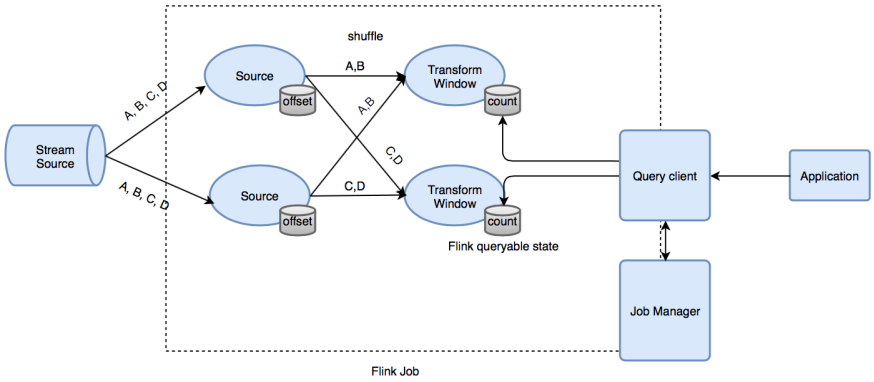

Figure 12: Stream processing with Queryable state[3]

Flink Queryable state addresses this issue by providing access to the application state in a Flink streaming job. Figure 12 represents how the architecture looks like with queryable state. Here the application uses the queryable client API, which asks the job manager for the task manager location where the actual state is held. Once the location information is available, actual requests are answered directly by the task manager that holds the state for the query key.

This is relatively a new feature in Flink (released in version 1.2.0) which includes an API to query the state from external applications. At the time of writing this thesis, queryable state is feasible only on keyed streams and manually managed state instances. That means, it is not yet possible to query the contents of a window (this is expected to be fixed soon). Also, the life cycle of queryable state is bound to the life cycle of a job. In future versions, the queryable state may be decoupled from the life cycle of a job allowing queries even after the job finishes.

5.1.1.3. RocksDB state backend

RocksDB [7] is a persistent key value store developed and maintained by Facebook Database Engineering Team, where keys and values are arbitrary byte streams. RocksDB provides extremely low latency and supports atomic reads and writes. RocksDB is designed in such a way that it is performant for fast storage and for server workloads. RocksDB provides flexible configuration settings that can be tuned for different production environments such as pure memory,

[3]adapted from: https://data-artisans.com/blog/queryable-state-use-case-demo

flash, hard disk or HDFS, and support high random-read workloads and storing terabytes of data in a single database.

RocksDB is one of the state backends supported by Flink out of the box. Flink encourages to use the RocksDB state backend for jobs with very large key/value state and where high availability is required. For RocksDB state backend, the size of the state is only limited by the amount of disk space available. Also RocksDB is the only state backend which supports incremental checkpointing, thus allowing to drastically reduce the checkpointing time compared to full checkpoints. Incremental checkpoints only record the changes from the previous completed checkpoint rather than producing a full back up of the state.

5.1.1.4. Machine Learning on Flink

Flink's dedicated support for iterative computations including bulk and delta iterations [27] makes it ideal for many scalable machine learning applications. The machine learning library of Flink, called FlinkML [2] is a relatively new effort in the Flink community, with a vision *to add scalable machine learning algorithms which are easy to use and with minimal glue code in end-to-end machine learning systems*.

Flink community is also actively thinking about having a serving layer on Flink, called Flink-MS [4] for various machine learning models potentially trained in Flink or any other platforms. Flink MS will try to address the requirements such as live model updating, latency guarantees, bandit frameworks (for competing model selection) and model versioning. At the time of writing this thesis, Flink-MS is only in the initial discussion phase.

5.1.2. Apache Kafka

Apache Kafka [36] is a fast, scalable and durable publish-subscribe messaging system with built-in partitioning, replication and fault tolerance. On a high level, Kafka architecture looks as shown in the Figure 13.

[4] https://docs.google.com/document/d/1CjWL9aLxPrKytKxUF5c3ohs0ickp0fdEXPsPYPEywsE

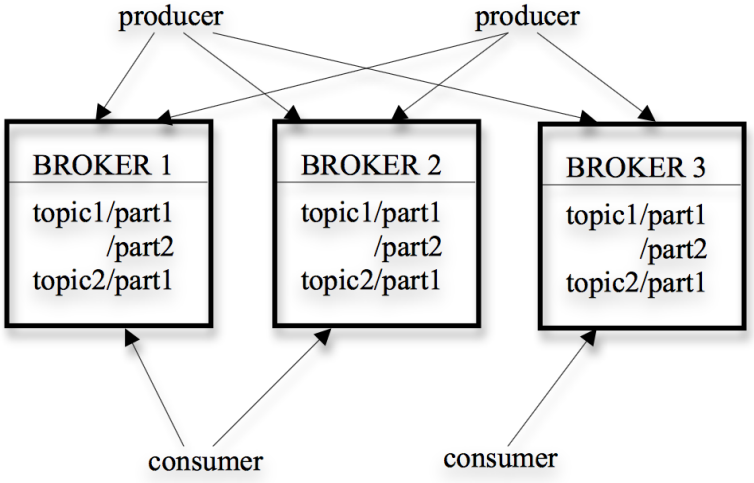

Figure 13: Kafka Architecture [36]

Kafka stores messages (also called as log) in topics that are partitioned and replicated across multiple brokers in a cluster. Producers read data from actual data sources and send them to Kafka topics. Kafka messages are byte arrays, and all the messages for a particular key are ensured to be in the same partition. Kafka can store the messages reliably for a finite time period. One or more consumers read the messages from the Kafka topic in this period. Multiple consumers may be grouped into a consumer group. Each message is delivered to one consumer in the group, and all the messages with the same key will arrive at the same consumer. Kafka keeps a per consumer offset (position of the consumer in the log) in order to identify which messages should be read next by the consumer. This offset is actually controlled by the consumer, making it feasible for the consumer to read the records in any order it likes. Normally a consumer will advance its offset linearly as it reads records.

Kafka provides persistent messaging using disk structures which scale well. That means Kafka will perform the same for TeraBytes of data as it will perform for KiloBytes of data. Even with modest hardware Kafka can provide high throughput, supporting hundreds of thousands of messages per second. Kafka has explicit support for partitioning the topics over Kafka server machines and distributing consumption over a cluster of consumer machines (for example, a distributed Flink job) while maintaining per partition ordering semantics. Also, for a topic with N as the replication factor, Kafka will tolerate N-1 server failures, providing fault tolerant message storage. All these features make Apache Kafka an ideal choice for stream source or sinks in distributed stream processing using platforms such as Apache Flink.

5.1.3. HDFS

HDFS [43] is the data layer in Apache Hadoop. It is a Java based file system which provides distributed, scalable and fault tolerant storage for very large data sets. HDFS is designed to be deployed on low cost hardware and provides high throughput access to the large data sets stored on it. The high level architecture of HDFS is shown in the Figure 14.

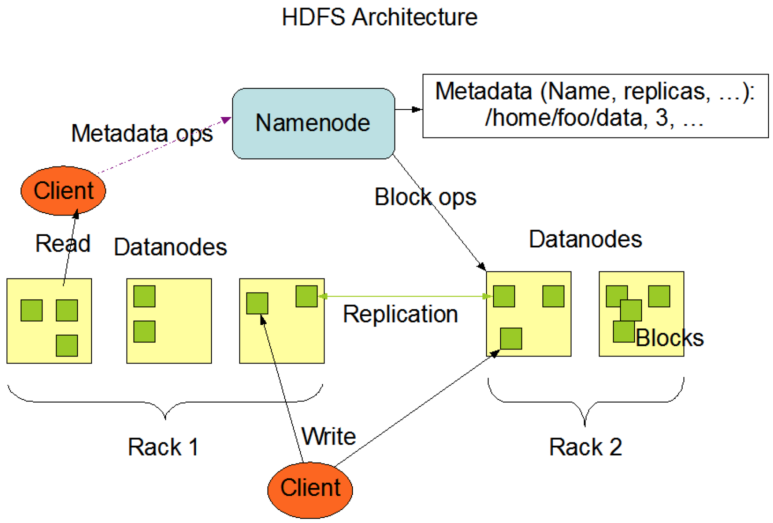

Figure 14: HDFS Architecture [5]

An HDFS cluster comprises of a NameNode which manages the cluster metadata, and DataNodes which stores the actual data. The data files are split into blocks (by default 128 MB). Each block will be replicated at multiple DataNodes and are stored on the local file system of the DataNodes. In the event of a DataNode failure, NameNode will create another replica of all the blocks stored on the failed node. When a client tries to read or write data to HDFS, NameNode acts as an orchestrator between the client and DataNodes, by providing the block location information to the client, where it can read/write the data. Client then directly contact the DataNode to fulfill the request. The entire archtirectue of HDFS is in such a way that no user data flows through NameNode. NameNode acts as constant monitor to serve the client requests, to manage node failures by replicating the data and it also ensures that data is balanced across the DataNodes in the cluster.

5.2. Architecture

In the Section 5.1 we discussed about an overview of the features offered by Apache Flink, Apache Kafka and HDFS. We discussed about real-time stateful stream processing offered by Flink, and how its queryable state feature allows to expose the state of the streaming job for external applications. We discussed about the reliable message queue features of Apache Kafka and how HDFS supports storing very large data sets on low cost hardware.

In this section we present the architecture of our solution for serving large machine learning models making use of the above mentioned technologies and the features offered by them. We propose a modular architecture which is extensible for various large scale machine learning models.

5.2.1. High level Architecture

The high level architecture of the solution is shown in the Figure 15. The primary components in the solution are Data Layer, Model Serving Layer and Batch Training Layer which is an optional component. In this section we discuss the high level functionalities of each of these components.

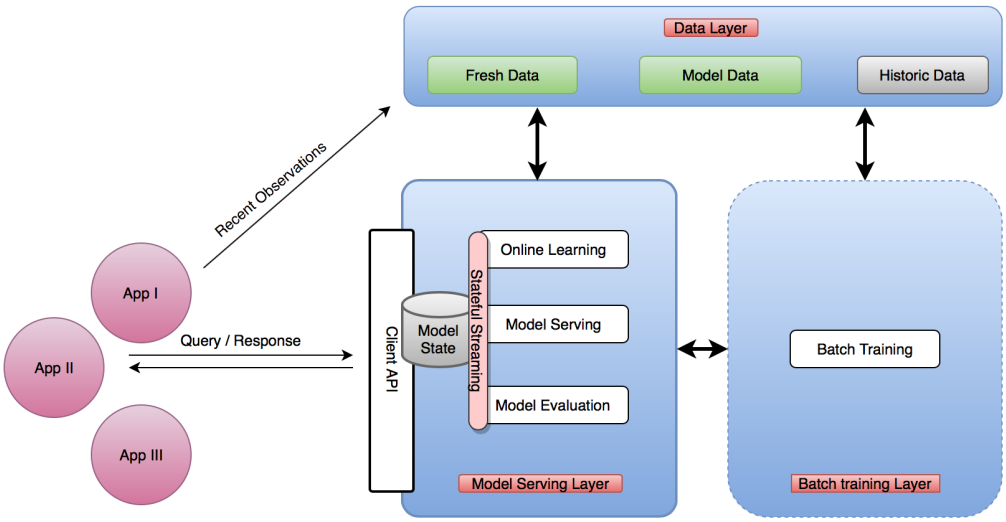

Figure 15: High Level Architecture

5.2.1.1. Data Layer

The data layer is responsible for handling three kinds of data:

(i) Freshly observed data from different applications, which is used for online updates to the model.
(ii) The actual trained model
(iii) Historical data which is used for batch training of the model.

The data layer should enable low latency access to the first two types of data, and batch access to the historical data. All three types of data should be available to the model serving layer and batch training layer in distributed and fault tolerant way. We use fast and reliable message queues offered by Apache Kafka to store fresh data and model data. The historical data is stored on the reliable low cost storage offered by HDFS. We explain the data layer design with Kafka and HDFS in detail in the Section 5.2.2. The choice of Kafka and HDFS is optional and can be replaced with other technologies which satisfies the features discussed above. For example, if your environment is Google cloud [8], the same can be replaced using Google Pub/Sub [6] and GFS [33].

5.2.1.2. Model Serving Layer

Model Serving layer forms the core component in our architecture, which handles the following vital functionalities.

1. Serving external applications: Model serving layer facilitates real-time predictions using the model in consideration, for different end-user applications.

2. Online learning: This component is responsible for updating the model based on the newly observed data in near real-time.

3. Model quality evaluation: This component evaluates the model either continuously or periodically and trigger the batch retraining if the model quality falls below a defined threshold.

All the three components demand fault tolerant and distributed architecture to serve large models. This is achieved using the stateful real-time stream processing and exposing the state of the applications for external queries. We chose Apache Flink with RocksDB state backend in order to implement Model Serving layer, as it satisfies all these requirements out of the box (See the Section 5.1.1 for the details of the state management and queryable state features of Flink). The details of Model Serving layer using Apache Flink are provided in the Section 5.2.2.

Any real-time, distributed and scalable stream processing system which offers exposing the state for external queries can replace Apache Flink in this layer. For example, Apache Samza [16] offers real-time and stateful stream processing, and it support using an external key value store as the state backend for making the state accessible for external applications. Using the external key value store certainly brings some bottleneck in the network. Samza is shipped with a key-value store implementation of LevelDB [10] using JNI API as a state backend, but at the time of writing this thesis, it is not yet exposed for external queries. Similarly, Apache Spark supports stateful stream processing based on micro-batching, but the state is saved in memory, and state backend is not configurable to support very large state for deploying and querying large models as per our design.

5.2.1.3. Batch Training Layer

Batch training layer facilitates training large models on the whole historical data in affordable time and cost. In our architecture, we included it as an optional component, meaning users can either use their existing model training infrastructure for training the models, or use this optional component which facilitates the model training as part of the model serving architecture. We use Apache Flink and its machine learning library called Flink ML as the batch training layer. The initial model is trained on the historical data in the data layer, and is deployed using model serving layer. Afterwards, model evaluation component in the model serving layer trigger the batch retraining and update the deployed model. Batch training layer can be implemented using any large scale distributed processing platforms, ideally with a rich machine learning Library, such as Apache Spark[46] with MLlib[42].

5.2.2. System Architecture

We discussed the high level functionalities of different components in our architecture in the Section 5.2.1. In this section we describe how these functionalities are satisfied in our system architecture using Apache Flink, Apache Kafka and HDFS. The Figure 16 shows the detailed system architecture with the technologies used for each components.

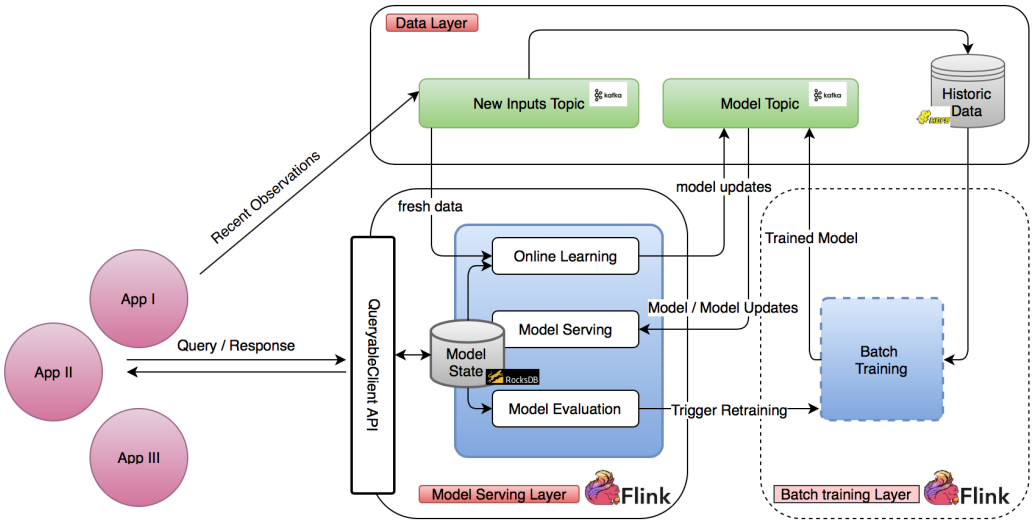

Figure 16: System Architecture

In order to understand the architecture comprehensively, we show the different steps involved in deploying a model on this architecture in the Figure 17. These steps are explained in detail below.

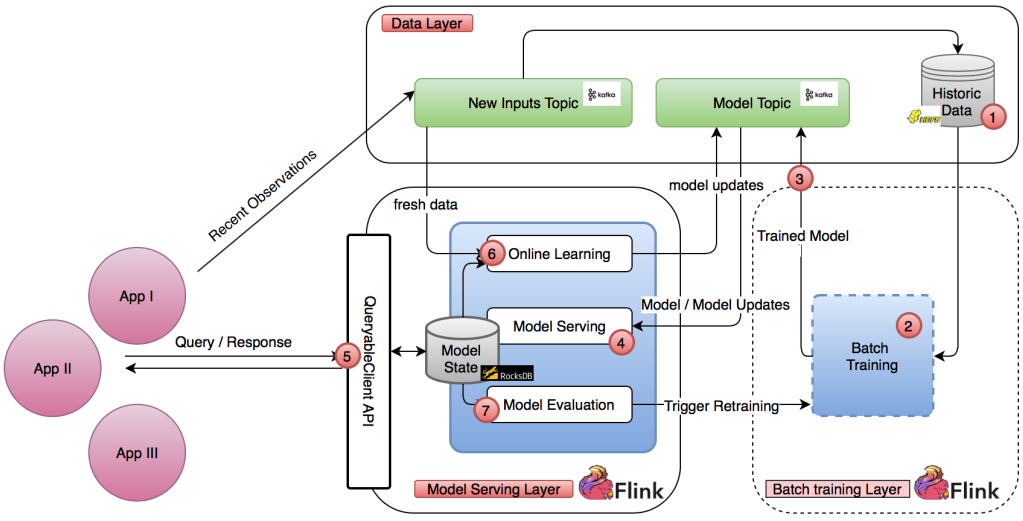

Figure 17: System architecture – Steps for deploying a model

1. **Collecting the data**: In the first step, we load the historical data to HDFS. Fresh data from various applications arrive at the Kafka new inputs topic, and is used to update the historical data periodically. This could be achieved using a Flink job which periodically read the Kafka topic and update the historical data on HDFS.

2. **Batch training**: The next step is to train the model using the historical data in our batch training layer. As we discussed in the Section 5.2.1.3, batch training layer may reside outside of the serving system.

3. **Loading model topic**: The trained model is loaded to a Kafka model topic. In our architecture we use Flink ML for training the model, and the resultant model is first stored to HDFS, and loaded to Kafka from HDFS using Flink streaming.

4. **Model serving with queryable state**: As the Kafka model topic is loaded initially, we deploy the Flink model serving job, which is a streaming application with RocksDB as the state backend. The streaming application load the model from Kafka topic to RocksDB state in such a way that low latency predictions are possible for different end user applications by querying this model. The streaming application then listens to model updates from Kafka topic, and update the RocksDB state accordingly.

5. **Querying the model**: As soon as the model serving job is deployed (step 4), end-user applications can query the model using the QueryableStateClient API [15] provided by Flink.

6. **Online learning**: Online model updates are performed as another Flink streaming application, which subscribes to the Kafka new inputs topic for fresh data, and reads the current model state using QueryableStateClient API. The model updates are then written to the Kafka model topic. As the model serving job (step 4) is subscribed to Kafka model topic, the model state in RocksDB also gets updated.

7. **Model Evaluation**: Model evaluation can be performed continuously or periodically. We have implemented model evaluation as a periodic job in Flink, which is executed on a test data set on HDFS and measure the quality of the model deployed in the RocksDB state. If the quality falls below a threshold value, a retraining of the entire model is triggered and as a result step 2 and 3 will be executed again.

5.3. Implementation

In this section we select two different machine learning models and explain the implementation of serving these models on our architecture in detail. We use Alternating Least Squares (ALS),

which is a collaborative filtering algorithm and Support Vector machines (SVM), which is a classification algorithm. We show how the same architecture is suitable for these different kinds of machine learning models.

We have implemented the solution in Java using the below software stack:

- Apache Flink 1.3.1
- Apache Kafka 0.10.2.0
- HDFS Hadoop 2.7.1

We discuss the end-to-end implementation of our solution for ALS model and show how to extend the architecture to other models by implementing the serving layer for SVM model. The online learning and model quality evaluation are not implemented for SVM model.

5.3.1. Alternating Least Squares (ALS)

In this section we explain the ALS [35] algorithm, and how ALS model serving is implemented on our architecture.

5.3.1.1. ALS Algorithm

ALS is a collaborative filtering algorithm for recommender systems based on matrix factorization. It factorizes the given matrix R into a user matrix U and an item matrix V, such that $R \approx U^T V$. The row dimension for the user and item matrix is given as a parameter to ALS, and is called latent factors. The i^{th} column of U is represented as u_i and the j^{th} column of V is represented as v_j. The matrix R is known as the ratings matrix with $(R)_{i,j} = r_{i,j}$

To find the user and item matrix, the below problem is solved

$$\underset{U,V}{\operatorname{argmin}} \sum_{\{i,j|r_{ij}\neq 0\}} \left(r_{ij} - {u_i}^T v_j\right)^2 + \lambda\left(\sum_i n_{u_i}||u_i||^2 + \sum_j n_{v_j}||v_j||^2\right) \tag{3}$$

where λ is called the regularization factor which is used to avoid over fitting. n_{u_i} is the number of items the user i has rated and n_{v_j} is the number of times the item j has been rated. ALS fixes one of the two matrices U and V, and the resultant quadratic equation can be solved directly. The solution decreases the overall cost function monotonically, and by applying this step alternatively to the user and item matrices, we can improve the factorization iteratively.

5.3.1.2. ALS Model Serving

in the Section 5.2.2, we discussed the different steps towards deploying a model on our architecture. In this section, we explain the details of these steps in deploying the ALS model. We provide the details of the Flink jobs involved, and the design and optimizations specific to the ALS model.

1. **Collecting the data**

   The historical data for training the model is stored on HDFS. In this implementation, we are also storing the fresh data on HDFS, and the online learning job which reads the fresh data, continuously monitor the HDFS directory for new data files. As we discussed in the section 5.2.1, the modular design allows us to plug-in different components. Using HDFS for fresh data assumes that a higher level system aggregates and store the fresh observations on HDFS. Where low latency is required, using Kafka for handling the fresh data is encouraged. While doing the batch training or retraining, the fresh data on HDFS is read along with historical data.

2. **Batch training**

   We use Flink ML implementation of ALS [1] for the batch training of the model. The implementation uses the below hyper parameters.

   i. **NumFactors**: Represents the number of latent factors to be used for training. A larger value in combination with very large number of users and items yield large models.

   ii. **Lambda**: Regularization Factor in order to prevent over-fitting.

   iii. **Iterations**: Number of iterations to perform while training the model

   iv. **Blocks**: represents the number of groups of users and items

   v. **Seed**: Random seed to generate the initial item matrix

   vi. **TemporaryPath**: The optional HDFS path where intermediate results should be stored.

   Flink ALS implementation allows the item matrix and user matrix after the training, to be stored separately on HDFS. We then load this model to Kafka topic as explained in the step 3. Figure 18 shows the execution plan of Flink ALS training job.

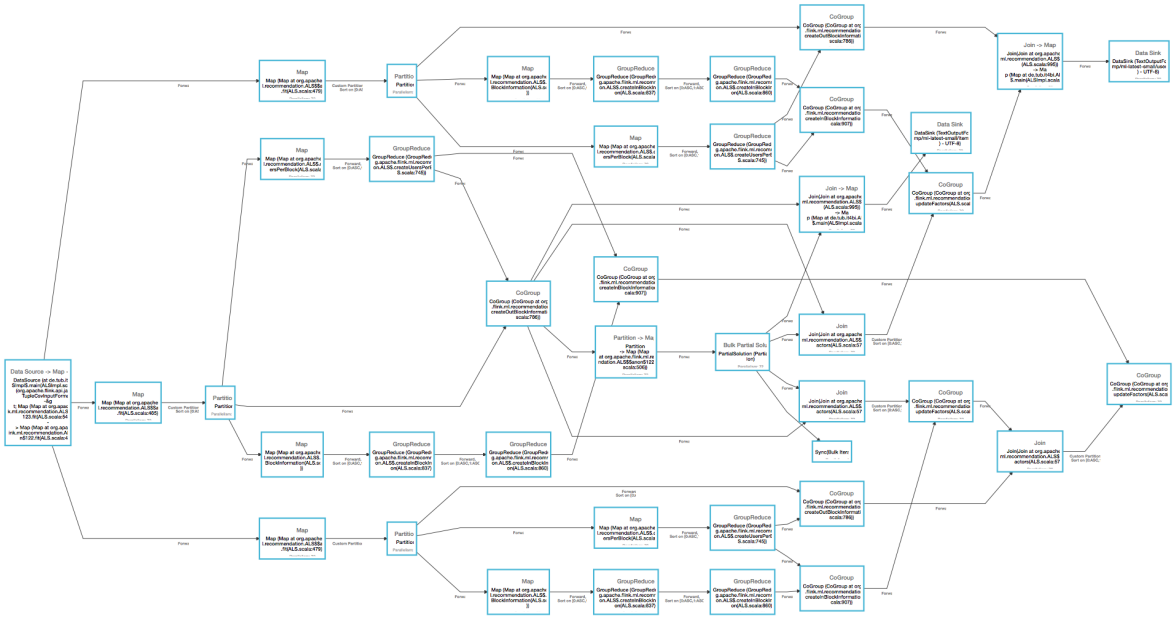

Figure 18: ALS Model Training – Execution Plan in Flink

3. **Loading model topic**

   The trained model (user and item vectors) on HDFS is loaded to Kafka model topic using a Flink batch job. This job is executed always after the batch training or retraining jobs. The execution plan of this job is as shown in the Figure 19.

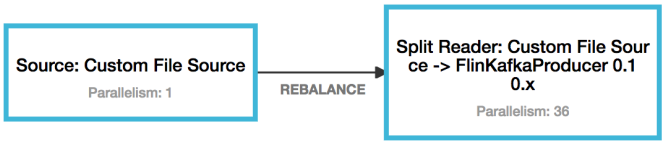

Figure 19: Execution Plan – loading the model to Kafka topic

4. **Model serving with queryable state**

   Once the Kafka topic is loaded with the trained model, this Flink streaming job is deployed, which reads the item and user vectors from Kafka model topic. This stream is stored as a value state in the RocksDB state backend with the userID or itemID as the key and the corresponding vector as the value. This value state is made queryable so that

external applications can access the model by the key. Checkpointing is enabled for the job so that in case of any failure, the job can restart from the last complete checkpoint. Figure 20 shows the execution plan for the model serving job.

Figure 20: Execution Plan – ALS model serving with queryable state

5. **Querying the model**

   Querying the model is made possible using the QueryableStateClient API [15] from Flink, which allows low latency queries to the model state in RocksDB state backend. In our implementation we use simple Java client to query the model. For serving multiple end-user applications, a RESTful API on top of the QueryableStateClient API may be used.

6. **Online learning**

   In online learning we address two aspects – how to update the ALS model for newly observed data, and how to model new users and items. We use Stochastic Gradient Descent (SGD) algorithm [25] in order to update the model for new observations. For each new rating data of the form <userID, itemID, rating>, we query the model in RocksDB to get the corresponding user vector and item vector and update them using SGD. The updated vectors are then sent to the Kafka model topic. Figure 21 shows the execution plan for SGD updates to ALS model.

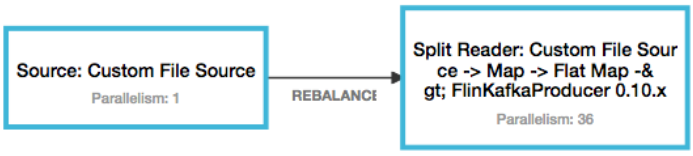

Figure 21: Execution Plan – online updates using SGD

For modeling the new users and items we calculate the mean latent factor vectors for users and items each time the batch training is carried out. This is achieved using a Flink job which reads the ALS model on HDFS, and calculate the mean factors and load them to the Kafka model topic. During online learning, if a user or item is not found in the model, these mean vectors will be queried and used in SGD step.

7. **Model Evaluation**

   For evaluating the model, we use mean squared error (MSE)[12] as the metric, and is implemented as a Flink job which can be executed periodically. The test data on HDFS is read and the error is calculated as the difference between the actual rating and the predicted rating. The output of model evaluation is the average of the squares of these errors. A smaller value of MSE (close to zero) indicate that the model quality is good. The automatic triggering of the batch training based on the MSE value is not implemented, instead it is handled manually. Figure 22 shows the execution plan of model evaluation job.

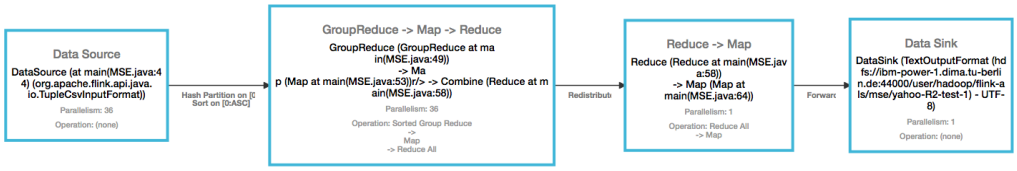

Figure 22: Execution Plan – Mean Squared Error calculation

5.3.2. Support Vector Machines(SVM)

In this section we explain the SVM [37] algorithm, and how SVM model serving is implemented on our architecture.

5.3.2.1. SVM Algorithm

Given the training data (x_i, y_i) for $i = 1...N$, with $x_i \in \mathbb{R}^d$ and $y_i \in \{-1, 1\}$, SVM learns a binary classifier such that

$$y_i = \begin{cases} +1, & \text{if } f(x_i) \geq 0. \\ -1, & \text{otherwise.} \end{cases} \tag{4}$$

SVM achieves this by creating a hyper-plane in a high dimensional space which classifies the training data into one of the two classes. This is illustrated in the Figure 23. Here the goal of the SVM is to select the hyper-plane $w.x + b = 0$, that maximizes the distance γ between the hyper-plane and any point in the training data.

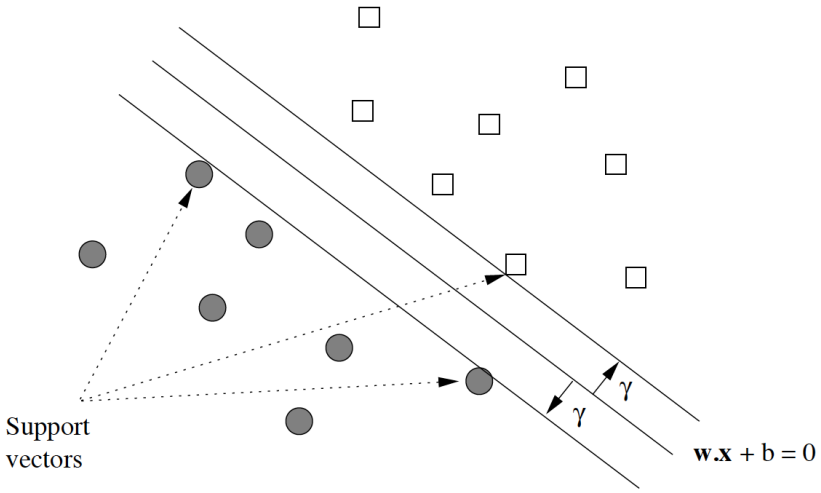

Figure 23: SVM mechanism [37]

The trained SVM model is an n-dimensional vector, where n is the number of features used in training the model. For a new observation, the class label is predicted by taking the dot product between the model vector and the new observation feature vector. Often the new observation will have values for a small number of features and it is enough to fetch the model parameters only for those features which are present in the observation (for the features which are not present in the observation, the dot product component will be zero). The class label is determined based on the dot product value and a predefined threshold value (i.e. if the dot product value is greater than the threshold, it belongs to positive class, and if it is less than the threshold, it belongs to the negative class).

5.3.2.2. SVM Model Serving

In this section we explain the details of serving SVM model on our architecture. We implement the batch training of the model using Flink ML implementation of SVM [3] and using data in LibSVM [30] format. The implementation takes the below hyper parameters to train the model.

i. **Blocks**: Sets the number of blocks into which the training data will be split. This should be at least equal to the degree of parallelism.

ii. **Iterations**: Number of iterations to perform while training the model

iii. **LocalIterations**: defines how many data points should be drawn from each local data block to calculate the stochastic dual coordinate ascent.

iv. **Regularization**: Defines the regularization constant of the SVM algorithm to avoid over-fitting

v. **Stepsize**: The initial step size for the updates of the weight vector. A larger step-size means larger contribution of the weight vector updates to the next weight vector value

vi. **ThresholdValue**: Defines the limiting value for the decision function. If the decision function value is above this threshold they are classified as positive class and if below, they are classified as negative class.

vii. **OutputDecisionFunction**: This is a Boolean value which represent if a binary class label (+1 and -1) should be returned or the actual distance of the support vector from the hyper plane should be returned as the output of the classfier.

viii. **Seed**: Defines the seed to initialize the random number generator

Figure 24 shows the execution plan for SVM model training on Flink. The trained model is an n-dimensional vector, where n is the total number of features. We partition this vector based on a user specified range. i.e. if the range is specified as 1000, each consecutive 1000 features will be stored as a single row in the output model.

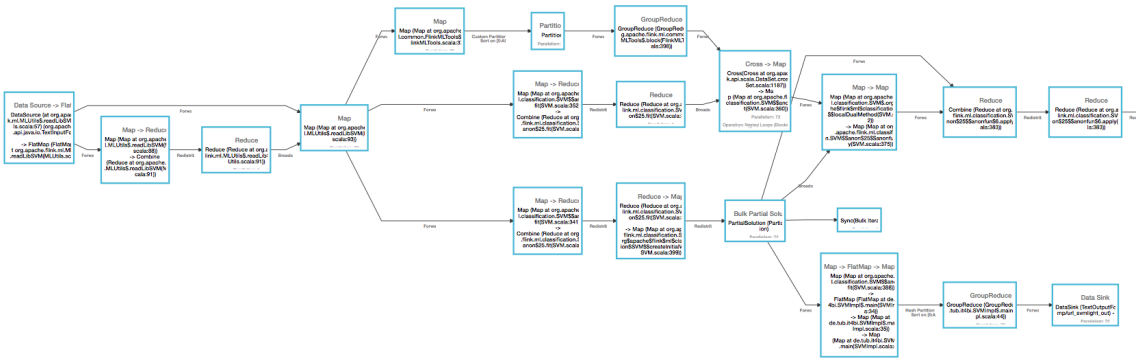

Figure 24: SVM Model Training – Execution Plan in Flink

This partitioned SVM model is then loaded to the Kafka model topic, and a Flink streaming job which subscribe to the Kafka model topic loads the model to queryable state using RocksDB similar to the ALS model implementation. Figure 25 shows the execution plan for SVM model serving job. The difference is that in ALS model, each user and item vectors were loaded to the RocksDB state and clients could query the model by user ID or item ID. In SVM model implementation, clients will inspect the features in the prediction request vector, group them to the ranges specified in the model training, and queries the model once for a particular range of keys.

Figure 25: Execution Plan – SVM model serving with queryable state

6. Evaluation

In this section, we present the evaluation of our solution with respect to the defined quality metrics. We discuss the performance and possible optimizations to our solution based on the experiment results. The section is divided as follows – first we define the quality metrics. Then we explain the experiment setup and data sets used for the experiments. Finally, we present the experiments and analysis of the results.

6.1. Quality Metrics

The quality dimensions and measures used in the evaluation are summarized in the Table 1.

Name	**D/M**	**Description**	**Applicable components**
Latency	M	Time required for a single prediction request	External Queries
Throughput	M	Number of predictions served in unit time	External Queries Online Learning Model Evaluation
Run-time	M	Time required to complete the operation	Model Loading
Number of Records	D	Total number of input records used to train/test the model	
Size of the model	D	The size of the trained model	
Number of features	D	Total number of features used in training the model	
Parallelism	D	Number of parallel tasks deployed in the operation	

Table 1: Quality Measures and Dimensions

No.	Item	Info	
1	JobManager	Architecture	ppc64
		No. of CPUs	48
		Memory	48G
		Total DiskSpace	1.9T
2	TaskManager	Architecture	ppc64
		No. of CPUs	48
		Memory	62G
		Total DiskSpace	4T
3	No. of Task Managers	3	
4	Flink Version	1.3.1	
5	Total No. of task slots	72	

Table 2: Details of the Flink Cluster

6.2. Experimental Setup

Table 2 shows the details of the Flink cluster setup and Table 3 shows that of the Kafka cluster. Tables 4 and 5 provide the explicit configurations used for Flink and Kafka respectively. Any other configuration which are not mentioned in these tables will take the default configuration values[5].

No.	Item	Info	
1	Kafka Node	Architecture	ppc64
		No. of CPUs	48
		Memory	62G
		Total DiskSpace	4T
2	No. of broker Nodes	3	
3	Kafka Version	0.10.2.0	

Table 3: Details of Kafka Cluster

[5]Default configrations can be found at:

1. Flink: https://ci.apache.org/projects/flink/flink-docs-release-1.3/setup/config.html
2. Kafka: https://kafka.apache.org/0101/documentation.html

Item	Value	Description
jobmanager.heap.mb	10240	The heap size for the JobManager JVM
taskmanager.heap.mb	61440	The heap size for the TaskManager JVM
taskmanager.numberOfTaskSlots	24	The number of task slots that each TaskManager offers. Each slot runs one parallel pipeline
taskmanager.memory.preallocate	FALSE	Specify whether TaskManager memory should be allocated when starting up (true) or when memory is required in the memory manager (false)
taskmanager.network.numberOfBuffers	167936	The number of buffers available to the network stack
query.server.enable	TRUE	Enable queryable state
query.server.port	2049	Port to bind queryable state server to
query.server.network-threads	0	Number of network (Netty's event loop) threads for queryable state server (0: picks number of slots).
query.server.query-threads	0	Number of query threads for queryable state server (0: picks number of slots)

Table 4: Flink Configuration

Item	Value	Description
zookeeper.properties		
maxClientCnxns	0	disable the per-ip limit on the number of connections
initLimit	5	timeouts ZooKeeper uses to limit the length of time the ZooKeeper servers in quorum have to connect to a leader
syncLimit	2	limits how far out of date a server can be from a leader
server.properties		
delete.topic.enable	TRUE	Switch to enable topic deletion or not
num.network.threads	3	The number of threads handling network requests
num.io.threads	8	The number of threads doing disk I/O
socket.send.buffer.bytes	102400	The send buffer (SO_SNDBUF) used by the socket server
socket.receive.buffer.bytes	102400	The receive buffer (SO_RCVBUF) used by the socket server
num.partitions	1	The default number of log partitions per topic
num.recovery.threads.per.data.dir	1	The number of threads per data directory to be used for log recovery at startup and flushing at shutdown
log.retention.hours	168	The minimum age of a log file to be eligible for deletion due to age
log.segment.bytes	1073741824	The maximum size of a log segment file. When this size is reached a new log segment will be created.
log.retention.check.interval.ms	300000	The interval at which log segments are checked to see if they can be deleted according to the retention policies
zookeeper.connection.timeout.ms	6000	Timeout in ms for connecting to zookeeper

Table 5: Kafka Configurations

6.3. Data Sets

We use Yahoo! R2 dataset [18] for experiments on ALS model serving and URL Reputation dataset [40] for experiments on SVM model serving.

6.3.1. Yahoo! R2 Dataset

This dataset is a snapshot of Yahoo! Music community's ratings for different songs. It contains more than 717 million ratings with 136 thousand songs and 1.8 million users. The total size of the dataset is 11G. Standard pre-processing for collaborative filtering and rating prediction was applied in order to create this dataset. The data is trimmed such that each user has rated at least 20 songs and each song is rated by at least 20 users. The ratings for each user have been randomly partitioned to training and test datasets. The test dataset consists of 10 ratings and the training dataset consist of the remaining ratings of each user. Each training data file consist of rating data of 200,000 users, with at least 10 ratings per user. The rating values are on a scale from 1 to 5. Each row in the data set is of the form: "user id<TAB>song id<TAB>rating".

6.3.2. URL Reputation Dataset

This dataset contains anonymized 120-day subset of the ICML-09 [40] URL data. It contains 2.4 million examples and a total of 3.2 million features. The data is in SVM-light format, with a class label +1 representing a malicious URL and -1 representing a benign URL.

6.4. Generated Models

In order to test the latency and throughput for very large models, we generate artificial models using Flink distributed jobs. For ALS model, the input parameters are number of users, number of items and the number of latent factors. The Figure 26 shows the execution plan for this job. The two map operators in the execution plan generates user vectors and item vectors respectively. Each vector consists of random generated values for the latent factors. A union of the generated user and item vectors is then performed to produce the final model.

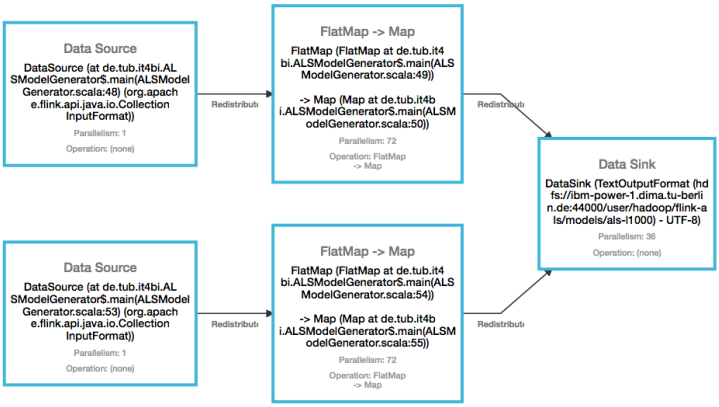

Figure 26: ALS Model generator – Execution Plan

For SVM model generator, the input parameters are the number of features, and the range (number of features to be stored in each partition). Figure 27 represents the execution plan for this job. The map operator generates the SVM model, with either random feasible values or zero as the feature values.

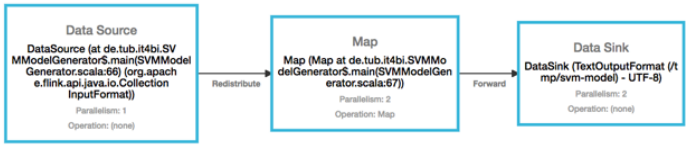

Figure 27: SVM Model generator – Execution Plan

6.5. Experiments and Results

In this section we present the experiments to measure the latency and throughput for different model settings with respect to the dimensions presented in the Table 1. We also analyze the runtimes for loading the trained model to the serving layer. The batch training layer is an optional component in our architecture. The solution can serve models trained on any other platform as well. Hence we do not perform any evaluation of this component. The experiments assume that the trained model is already available.

6.5.1. Latency

We measure the latency of a prediction request as the time taken to query the model parameters and to calculate the prediction using the retrieved parameters. In this section we present the latency experiments conducted on ALS and SVM model serving.

6.5.1.1. ALS model serving

In the case of ALS model serving, latency measures the sum of the time taken to retrieve the user and item vectors from model serving layer and to calculate the dot product of these vectors at the client application. Figure 28 shows the latency experiment results on ALS model serving for Yahoo! R2 dataset with 10 latent factors, and Figure 29 shows ALS model with 100 latent factors. In both cases, the average latency is in the range of 5 milliseconds. The initial spike in both curves shows that the initial query takes longer (in the range of 500 milliseconds). This is because the client communicates with JobManager to perform a location lookup to find out which TaskManager actually holds the model parameters for the query key. After this initial query, client caches it for later requests. Further requests are answered directly by the TaskManager that holds the model parameters for a specific key. Figure 30 shows the latencies for 1000 queries. Apart from the very few queries which shows latencies in the range of 50 milliseconds, majority of the queries shows consistent latencies in the range of 5 to 10 milliseconds.

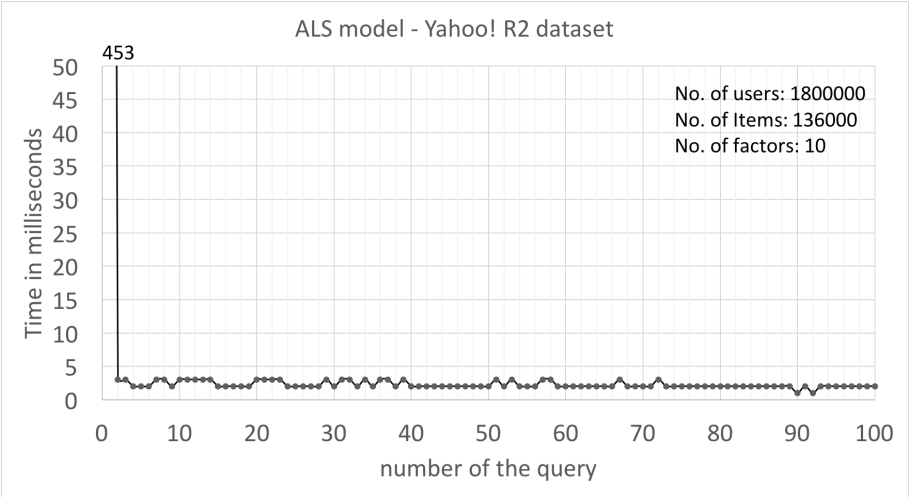

Figure 28: Latency: ALS model serving – Yahoo! R2 data (10 factors, 100 queries)

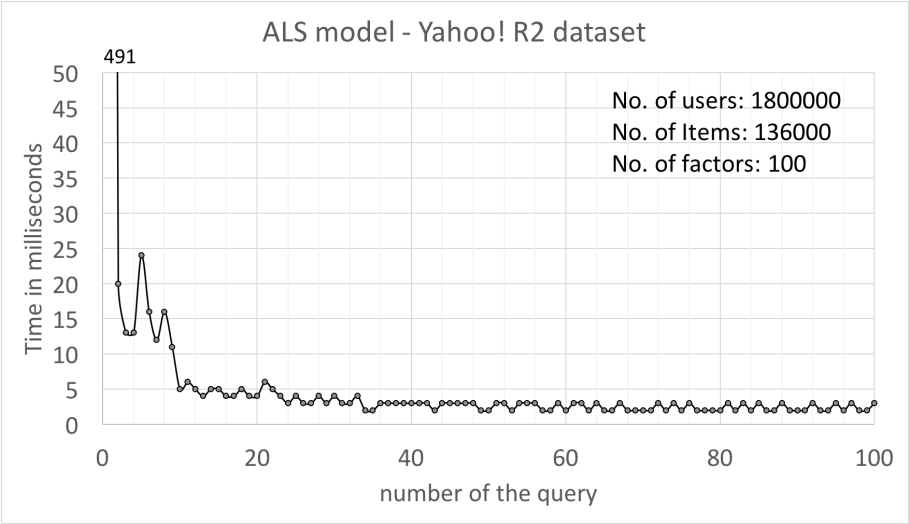

Figure 29: Latency: ALS model serving – Yahoo! R2 data (100 factors, 100 queries)

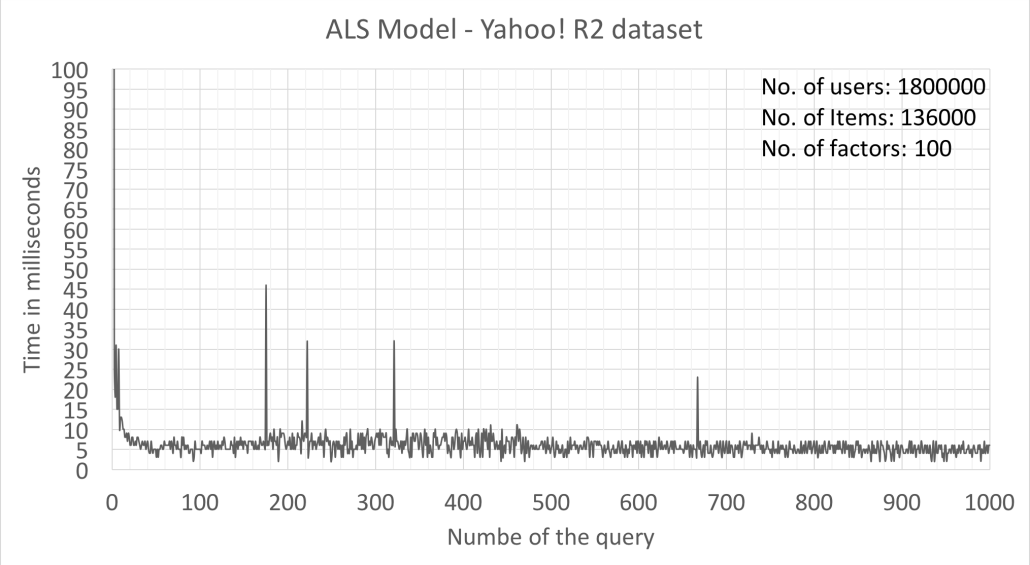

Figure 30: Latency: ALS model serving – Yahoo! R2 data (100 factors, 1000 queries)

Figure 31 shows the latencies for generated ALS models, where we use large values for latent factors. For all the generated models we use 2M users and 500,000 items. The resultant models are very large in size, and the plot shows acceptable latencies even for very large models. For latent factors 100 and 500, the average latency is in the range of 10 milliseconds. For latent fac-

tors 1000, we see varying latency with an average value of 160 milliseconds. This is primarily due to the expensive dot product in the prediction calculation at the client side.

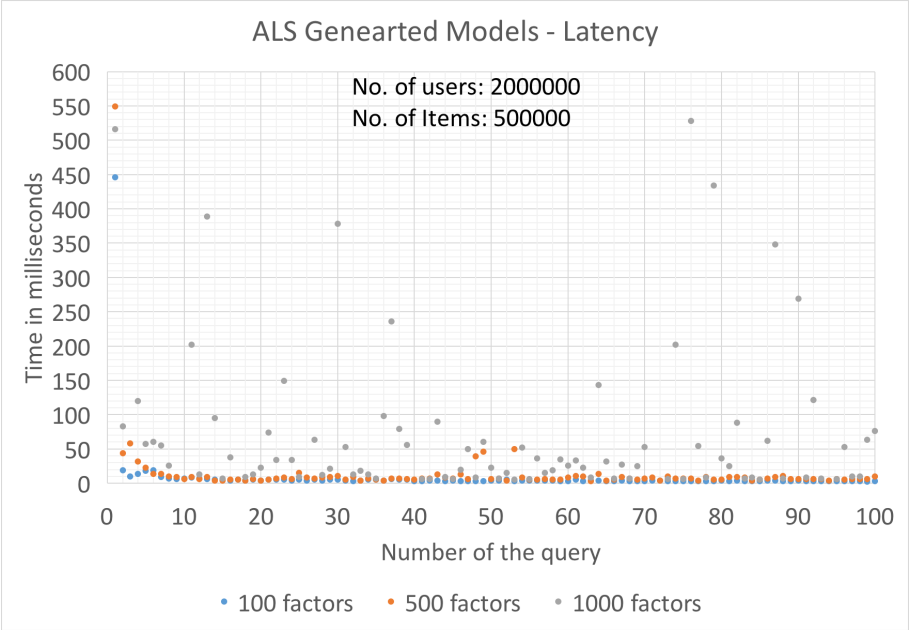

Figure 31: Latency: ALS model serving – Generated Models

6.5.1.2. SVM model serving

For SVM model serving, prediction latency measures the sum of the time taken to retrieve the feature values from the model and to calculate the dot product. We perform latency experiments on SVM model on URL Reputation dataset, which contains a total of 3231961 features. We partition these features into buckets of size 1000. For example, all the features from id 1001 to 2000 will be in bucket 2. The feature values for all the features in a bucket are stored in a single row in the model serving layer, with bucket ID as the key. The prediction requests are usually sparse and contain only a portion of the total features. At the client, we group the features in the prediction request to the same buckets used in the model serving layer. Then we query the model only once for all the features belonging to a particular bucket. In this way we reduce the total number of queries to the model.

We perform the latency experiment with random value of features (with a maximum value of 100,000) in the prediction requests. The Appendix A shows the number of features present in the prediction requests used in this experiment. Figure 32 shows the latencies for this model in the range of 300 to 400 milliseconds. This is expected as the model needs to be queried multiple

times in order to serve a single prediction request. The dot product is also expensive compared to ALS model.

In order to obtain lower latencies, we can partition the model based on the frequently accessed features. Generally, an SVM model will have certain features which will be present in most of the prediction requests whereas other features appear rarely. By partitioning these 'hot' features together, a few queries to the model allows to serve the prediction request with improved latency. This optimization is not implemented in our work.

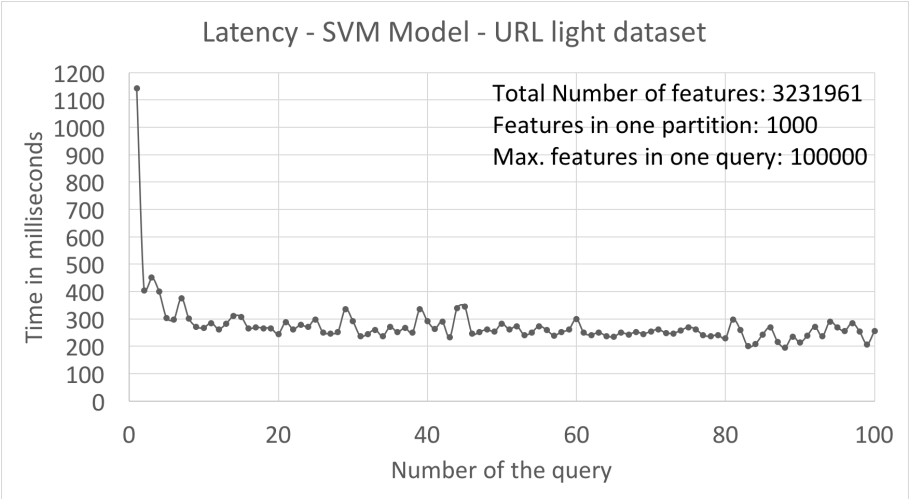

Figure 32: Latency: SVM model serving – URL Reputation data

6.5.2. Throughput

We measure the throughput using the below definitions.

1. For prediction requests from clients, throughput is the number of prediction requests served per second. This involves retrieving the model parameters from model serving layer and calculating the prediction at the client using the model parameters. The same applies to model quality evaluation job.

2. For Online learning, the throughput is measured as the number of records written to Kafka topic per second after the SGD step. During online learning, we also measure the throughput for model serving job as the number of records reaching the queryable state operator per second.

In this section we present the throughput experiments conducted on ALS and SVM model serving. For ALS model serving, we perform throughput experiments for both prediction requests and online learning. For SVM model serving throughput experiments are performed only for prediction requests.

6.5.2.1. ALS model serving

Prediction Requests

Figure 33 shows the throughput for prediction requests from a single client on ALS model using Yahoo! R2 data. The curve shows that the throughput increases as the number of queries increases. Then it becomes steady for larger number queries, yielding a high consistent throughput in the range of 700 predictions per second.

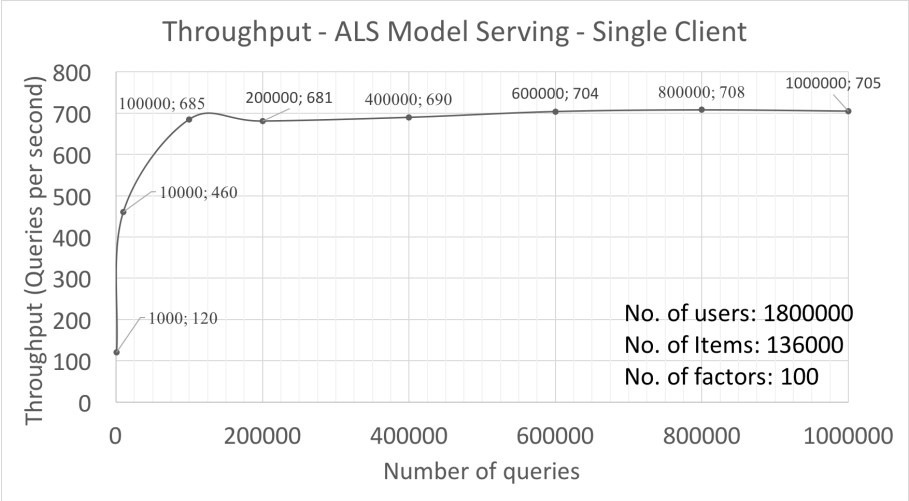

Figure 33: Throughput of a single client: ALS Model Serving – Yahoo! R2 data

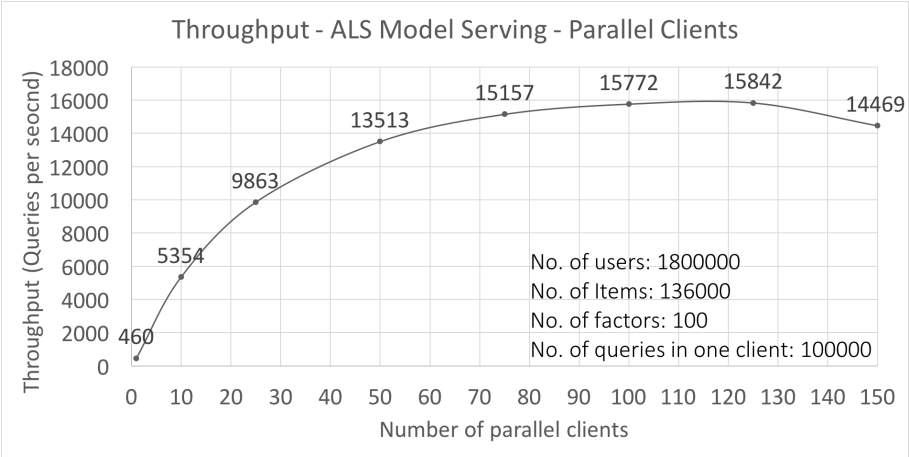

Figure 34: Throughput of parallel clients: ALS Model Serving – Yahoo! R2 data

For the same settings, Figure 34 shows the throughput for parallel clients. The throughput increases as the number of parallel clients increases. We achieved a maximum throughput of 15842 queries per second, when tried with 125 parallel clients. The throughput decreases for larger number of parallel clients. This is due to two reasons – i) For large number of concurrent queries, we observe time out exceptions in QueryableStateClient for small time outs. This shows that the queries take more time to process in such cases. ii) The parallel clients were run on the same machine. The throughput also measures the client side calculations involved in prediction request. For large parallel clients this becomes slow due to the decrease in available resources per client.

In order to serve large number of applications, it is ideal to implement a REST API, which acts as the gateway between model serving layer and external applications. The REST API internally implement one or more QueyableStateClient and redirect the prediction requests from external applications through these QueyableStateClient to the the actual model state in RocksDB. We can also cache the frequently accessed user and item vectors at the REST server or at the client applications so that queries to the model state is reduced. These optimizations are not implemented in this work.

We measure the throughput for very large model states using the generated ALS models (Figure 35). The generated models contain 2M users and 500,000 items. We plot the variation in throughput as the number of latent factors increases. The curve shows that throughput decreases linearly with increase in latent factors.

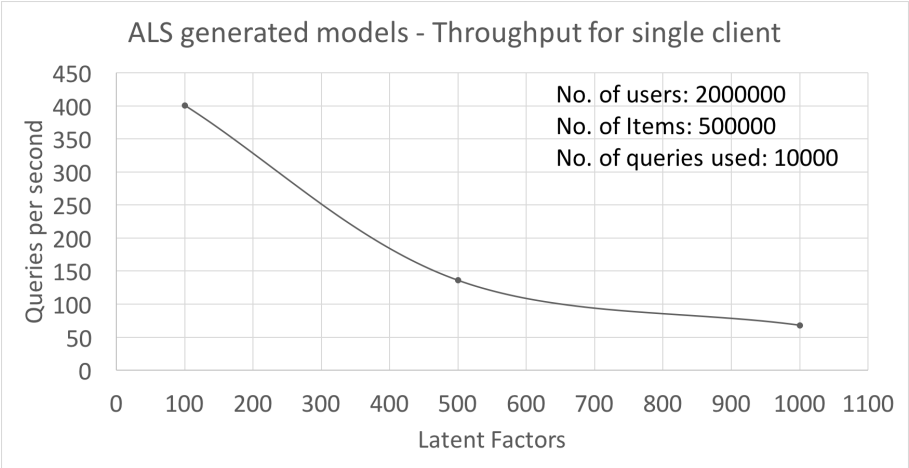

Figure 35: Throughput of single client: ALS Model Serving – Generated models

Model Quality Evaluation

We conducted the model quality evaluation on the ALS model trained on Yahoo! R2 train data. We used the Yahoo! R2 test data for calculating the MSE. For this job, the throughput is measured as the number of records produced per second in one of the operators in the job. This operator takes all the test data for a particular user together. It then queries the model for that user id and all the item ids in the test data for that user. For each rating in the test data, prediction is calculated as the dot product between user and item vectors retrieved from the model, and then it emits the prediction along with the actual rating to the subsequent operator. In this way, throughput of this operator is similar to an external prediction request.

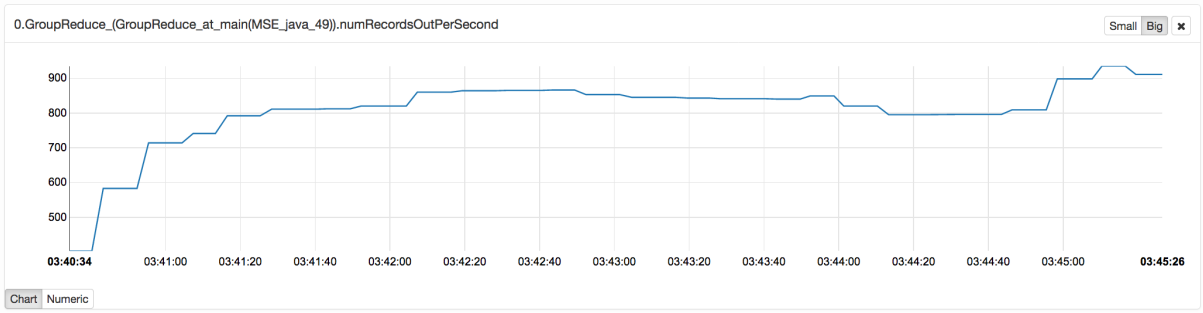

Figure 36: Throughput in Model Quality Evaluation: ALS model serving - Yahoo! R2 data

The Figure 36 shows Flink provided task metric which shows the number of records out per second from this operator for a time window of 5 minutes. The operator throughput averages at 800 records per second. This is slightly more than the throughput for prediction request

from external clients. Each prediction in this operator involves querying only the item vector compared to querying both user and item vectors for the prediction requests from external client applications. The user vector here is queried only once and cached for further predictions.

This is the throughput per operator. If the Flink job is deployed with a parallelism of 10, the total throughput will be 10 times the throughput of this single operator. In our experiment we used a parallelism of 36, thus yielding a total throughput of 25,200 records per second.

Online learning

We use the ALS model trained on Yahoo! R2 train dataset to conduct the online learning experiment. The ratings in the Yahoo! R2 test dataset is used as the fresh observations. For the online learning job, we measure the throughput at two instances – i) The Operator which writes the model updates to Kafka topic. Here we measure the number of records per second being written to Kafka topic. ii) While the online learning job is running, the model serving job listens to the updates in Kafka topic and updates the model state in RocksDB continuously. Here we measure the number of records reaching the queryable state per second.

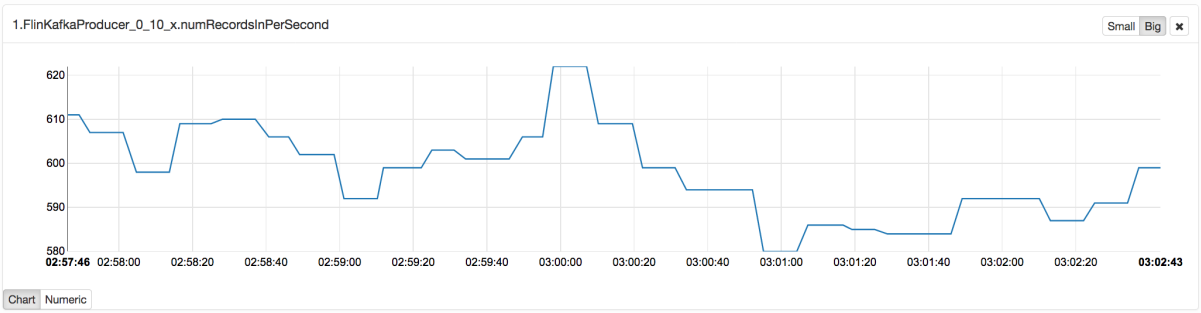

Figure 37: Throughput in online learning - Kafka updates: ALS model - Yahoo! R2 data

Figure 37 shows the throughput in writing the model updates to Kafka topic. The average throughput here is 600 records per second per operator (The 5-minute window shows a minimum value of 580 and a maximum value of 620 records). We deployed the online learning job with 36 parallel tasks, yielding a total throughput of 21600 records per second. Figure 38 shows the throughput at which model updates reaches the queryable state operator. The throughput here is less than the throughput for updating the Kafka topic. It averages at 430 records per second per operator (The 5-minute window shows a maximum value of 445 and minimum of 420 records).

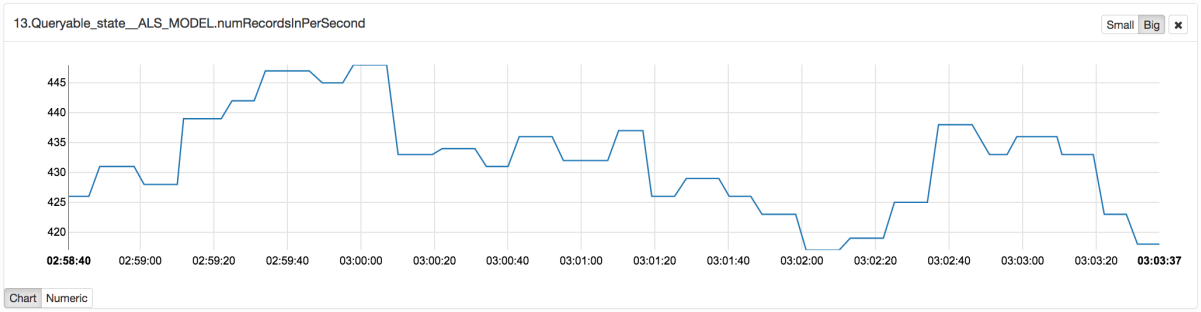

Figure 38: Throughput in online learning - model state updates: ALS model - Yahoo! R2 data

In this implementation, we update the user vector and item vector for each new observation and update the model state immediately. Parallel online learning tasks which work on the same user or item vectors will try to update the model simultaneously and RocksDB can handle these update requests. However, for large number of online updates, we can also perform some optimizations. We can use a time window for the SGD operator. All the training data which reaches this window will query the model for the user and item vectors and will calculate the gradients. At the end of the time window, it can emit the gradients to an aggregator operator. It will aggregate the gradients of the same user or item ID coming from different parallel tasks of the window operator. Updated user and item vectors are then written to the model state. In this way, number of updates to the model can be reduced significantly at the cost of slightly delayed updates to the model.

The number of queries to the model can also be optimized by grouping the new training data based on the user or item ID, so that all the records for a particular key will go to the same operator. For example, Yahoo! R2 dataset contains very large number of user IDs compared to item IDs. In this case, grouping the online training data on user ID allows us to query each user ID only once in the SGD operator. The grouping ensures that all the training data for this user will arrive only at this operator. In this case we can even calculate the gradient for the user ID completely in one operator. Item gradients can be sent downstream to aggregate as explained before.

6.5.2.2. SVM Model Serving

Figure 39 shows the throughput for SVM model on URL Reputation dataset. For prediction requests with a maximum of 100,000 features, we observe an average throughput of 4 queries per second. The throughput here is primarily limited by the number of features present in the prediction request. Figure 40 shows that the throughput remains more or less the same even for larger models. SVM model on URL Reputation dataset holds 3231961 features. The generated

models shown in Figure 40 contain 100000000 total number of features. The plot shows that holding more features in one row in the model and thus reducing the queries to the model improves the throughput. The difference is significant in the beginning. For higher values the difference is negligible. As we discussed in the Section 6.5.1.2, partitioning frequently accessed features together will yield higher throughputs.

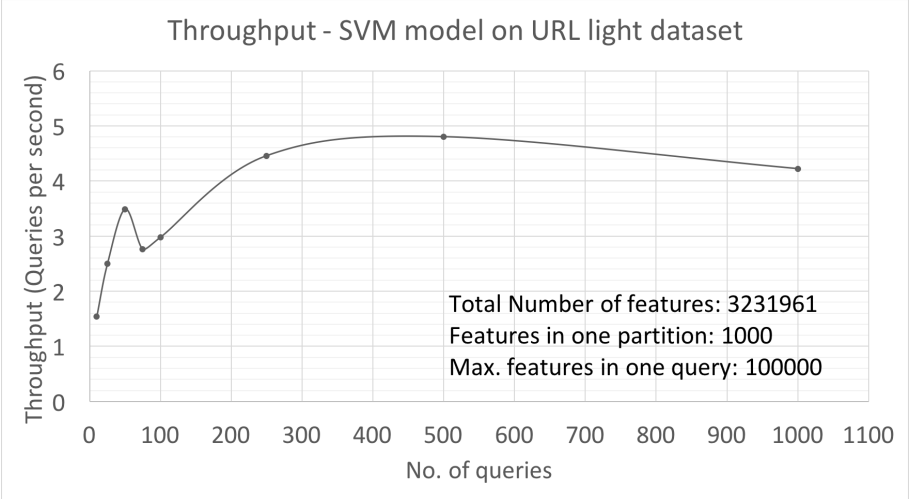

Figure 39: Throughput: SVM Model Serving – URL Reputation data

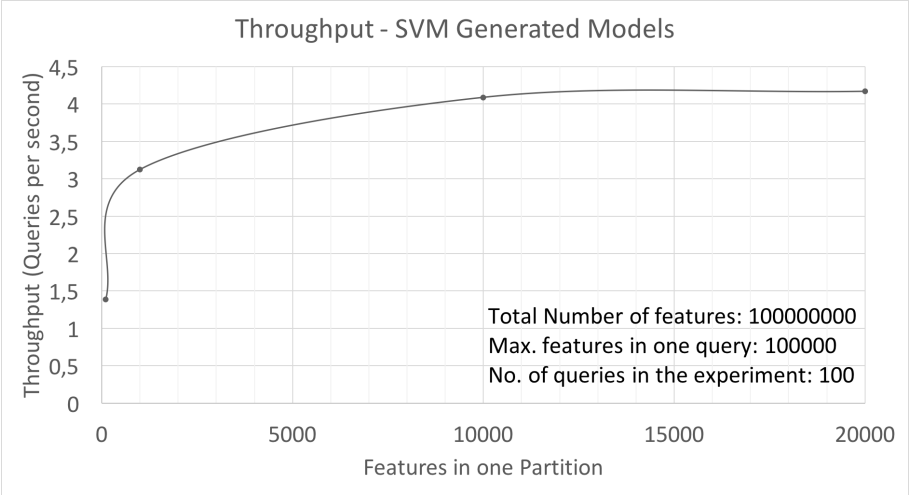

Figure 40: Throughput: SVM Model Serving – Generated models

6.5.3. Run-times

In this section we discuss the run-times for loading models of different sizes to the serving layer. We show the results on ALS model serving in the Figure 41. The run-time includes the time taken for the model loading job to Kafka topic and the time taken to reflect the model in RocksDB state. Both jobs run in parallel. As we discussed in the Section 6.5.2.1, the throughput for updating queryable state is less than the model loading to Kafka topic. Proportional to these throughputs, the model loading to RocksDB state takes more time than the model loading to Kafka topic. The run-times increase linearly with the size of the models. We divided the total number of task slots in the Flink cluster equally to these two jobs. As both jobs are distributed, increasing the task slots (i.e. adding more nodes to the Flink cluster) will also result in better run-times. Having large number of partitions for the Kafka topic also helps in improving the run-times.

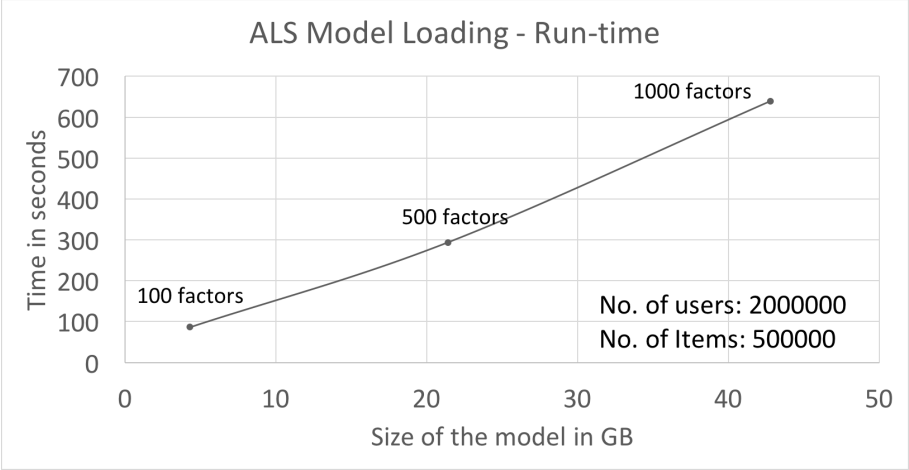

Figure 41: Run-times: Loading generated ALS Models

6.5.4. Online Learning and Model Quality

We performed online learning on the ALS model using Yahoo! R2 dataset. The dataset is divided into a training set of 699640226 ratings and a test set of 18231790 ratings. We use the training set to produce the ALS model using 100 latent factors and is loaded to the serving layer. We estimated the mean squared error(MSE) of this model using the model quality evaluation module. The model showed an MSE value of 2.46 (The high value is because this is not an optimized model as we didn't perform tuning experiments for the latent factors). We then performed online learning using SGD on this model using the test dataset. We noticed that a total of 36,463,580 updates were performed on the model during this online job, which is exactly

double the amount of ratings present in the test set. This is because, for each new observation in the test data, both user and item vectors will be updated in the ALS model. On our Flink cluster with 36 parallel tasks, this job finished in 49.7 minutes. We have already discussed about the throughput of this job in the section 6.5.2.1. We evaluated the model quality again, and the MSE value reduced from 2.46 to 1.97. This experiment shows that the quality of the model can be improved using online learning.

7. Conclusion

We have developed a novel model serving solution for large machine learning models using the real-time stateful stream processing and queryable state features offered by Apache Flink [27] and reliable message queues offered by Apache Kafka [36]. With the RocksDB [7] state backend for queryable state, we are able to serve low latency and high throughput predictions to end-user applications.

Both Flink and Kafka are scalable and fault tolerant, which makes our serving solution robust and allows quick deployment of very large models. Our solution provides online updates to the deployed model as the fresh data arrives. The solution also supports continuous evaluation of the model quality and trigger periodic retraining of the model if quality falls below a defined threshold. Moreover, models trained on any platform can be served using our solution.

We provided a detailed survey of the existing model serving frameworks and saw that existing solutions do not satisfy all the required features for model serving. Clipper [32] and Oryx2 [13] does not address serving large models. LASER [21] does not support deploying different types of models and Tesnsorflow Serving [9] supports only Tensorflow models out of the box. PredictionIO [28] does not provide online learning out of the box and Velox [31] supports only partial online updates to the model. Our solution provides all these features, and with the modular architecture, it is also possible to replace Flink and Kafka with other technologies which offer similar features.

We implemented our solution end-to-end (model serving, online learning and model quality evaluation) for large matrix factorization models using Alternating Least Squares (ALS) algorithm [35]. We showed how to extend our solution to other machine learning models by implementing the model serving on large classification models using Support Vector Machines (SVM) algorithm [37]. Our experiments on SVM models using URL Reputation dataset [40] showed latency in the range of 0.3 seconds. This high latency is due to the fact that SVM model prediction requires large number of queries to the model. Our experiments on ALS model using Yahoo! R2 dataset [18] demonstrated very low latency (less than 10 milliseconds), and high throughput. The online learning experiment showed that model quality was improved, with

mean squared error reducing from 2.46 to 1.97. With artificial ALS models of large sizes, we also demonstrated quick deployment of models in our serving solution.

8. Future Work

The next logical step of our work is to extend the model serving for different machine learning models. The models should be added as customizable model templates similar to PredictionIO [28]. The serving layer should allow to switch between multiple versions of the same model as well. Our architecture can fit these requirements easily.

The model quality evaluation is periodic in the current implementation. The next step in this regard is to have continuous prequential evaluation of the deployed models. For online learning, currently we update the model for each new observation. This should be customizable depending on the application. Users must be able to choose desired consistency levels for model updates.

Our implementation of ALS recommender model is complete except how to handle top-k recommendations. SVM model implementation needs optimizations such that the partitioning of the model features should consider the frequency of distribution of features in the training dataset. This will help to improve latency of serving prediction requests. Currently, we have provided a naïve implementation where sequential features are partitioned together. The future work should address these limitations in the current implementation of ALS and SVM models.

In order to serve large number of applications, the solution should provide a RESTful API which acts as the gateway between model serving layer and external applications. The RESTful API internally will manage one or more QueryableStateClient [15] in order to serve the prediction requests. Caching of the frequently accessed model parameters at this API layer will reduce the queries to actual model state, resulting in improved latency and higher throughput.

References

[1] Apache flink 1.3 documentation: Alternating least squares. https://ci.apache.org/projects/flink/flink-docs-release-1.3/dev/libs/ml/als.html. (Accessed on 08/13/2017).

[2] Apache flink 1.3 documentation: Flinkml - machine learning for flink. https://ci.apache.org/projects/flink/flink-docs-release-1.3/dev/libs/ml/. (Accessed on 08/13/2017).

[3] Apache flink 1.3 documentation: Svm using cocoa. https://ci.apache.org/projects/flink/flink-docs-release-1.3/dev/libs/ml/svm.html. (Accessed on 08/13/2017).

[4] Apache flink: Scalable stream and batch data processing. https://flink.apache.org/. (Accessed on 08/13/2017).

[5] Apache hadoop 3.0.0-alpha4 – hdfs architecture. http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html. (Accessed on 08/13/2017).

[6] Cloud pub/sub - message-oriented middleware — google cloud platform. https://cloud.google.com/pubsub/. (Accessed on 08/13/2017).

[7] Getting started — rocksdb. http://rocksdb.org/docs/getting-started.html. (Accessed on 08/13/2017).

[8] Google cloud computing, hosting services & apis — google cloud platform. https://cloud.google.com/. (Accessed on 08/13/2017).

[9] Introduction — tensorflow. https://www.tensorflow.org/serving/. (Accessed on 08/13/2017).

[10] Leveldb.org. http://leveldb.org/. (Accessed on 08/13/2017).

[11] Machine learning - wikipedia. https://en.wikipedia.org/wiki/Machine_learning. (Accessed on 08/13/2017).

[12] Mean squared error - wikipedia. https://en.wikipedia.org/wiki/Mean_squared_error. (Accessed on 08/13/2017).

[13] Oryx – overview. http://oryx.io/index.html#lambda_tier_implementation. (Accessed on 08/13/2017).

[14] prediction-serving-systems-cs294-rise_seminar. https://ucbrise.github.io/cs294-rise-fa16/assets/slides/prediction-serving-systems-cs294-RISE_seminar.pdf. (Accessed on

08/13/2017).

[15] Queryablestateclient (flink 1.3-snapshot api). https://ci.apache.org/projects/flink/flink-docs-release-1.3/api/java/org/apache/flink/runtime/query/QueryableStateClient.html. (Accessed on 08/13/2017).

[16] Samza. http://samza.apache.org/. (Accessed on 08/13/2017).

[17] Software — amplab – uc berkeley. https://amplab.cs.berkeley.edu/software/. (Accessed on 08/13/2017).

[18] Webscope — yahoo labs. https://webscope.sandbox.yahoo.com/catalog.php?datatype=r. (Accessed on 08/13/2017).

[19] Welcome to apache predictionio (incubating)! http://predictionio.incubator.apache.org/. (Accessed on 08/13/2017).

[20] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[21] Deepak Agarwal, Bo Long, Jonathan Traupman, Doris Xin, and Liang Zhang. Laser: A scalable response prediction platform for online advertising. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pages 173–182. ACM, 2014.

[22] Amr Ahmed, Moahmed Aly, Joseph Gonzalez, Shravan Narayanamurthy, and Alexander J Smola. Scalable inference in latent variable models. In *Proceedings of the fifth ACM international conference on Web search and data mining*, pages 123–132. ACM, 2012.

[23] Charles Anderson. Docker [software engineering]. *IEEE Software*, 32(3):102–c3, 2015.

[24] Taiwo Oladipupo Ayodele. Types of machine learning algorithms. In *New advances in machine learning*. InTech, 2010.

[25] Léon Bottou. Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade*, pages 421–436. Springer, 2012.

[26] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. Lightweight asynchronous snapshots for distributed dataflows. *arXiv preprint arXiv:1506.08603*, 2015.

[27] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[28] Simon Chan, Thomas Stone, Kit Pang Szeto, and Ka Hou Chan. Predictionio: a distributed machine learning server for practical software development. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 2493–2496. ACM, 2013.

[29] K Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[30] Chih-Chung Chang and Chih-Jen Lin. Libsvm: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):27, 2011.

[31] Daniel Crankshaw, Peter Bailis, Joseph E Gonzalez, Haoyuan Li, Zhao Zhang, Michael J Franklin, Ali Ghodsi, and Michael I Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. *arXiv preprint arXiv:1409.3809*, 2014.

[32] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *NSDI*, pages 613–627, 2017.

[33] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[34] M Levent Koc and Christopher Ré. Incrementally maintaining classification using an rdbms. *Proceedings of the VLDB Endowment*, 4(5):302–313, 2011.

[35] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8), 2009.

[36] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

[37] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge university press, 2014.

[38] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–15. ACM, 2014.

[39] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *OSDI*, volume 1, page 3, 2014.

[40] Justin Ma, Lawrence K Saul, Stefan Savage, and Geoffrey M Voelker. Identifying suspicious urls: an application of large-scale online learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 681–688. ACM, 2009.

[41] H Brendan McMahan, Gary Holt, David Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, Daniel Golovin, et al. Ad click prediction: a view from the trenches. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1222–1230. ACM, 2013.

[42] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[43] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. IEEE, 2010.

[44] Alexander Smola and Shravan Narayanamurthy. An architecture for parallel topic models. *Proceedings of the VLDB Endowment*, 3(1-2):703–710, 2010.

[45] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[46] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

Appendix

A. Appendix A

A.1. No. of features in Latency Experiments: SVM Model

The table 6 shows the number of features present in the prediction requests used in the latency experiments on SVM model (described in the section 6.5.1.2). The table shows the number of the query in the experiment, number of features present in the query and time taken to process the query in milliseconds.

No.	Features	Time	No.	Features	Time	No.	Features	Time	No.	Features	Time
1	59656	1143	26	30932	250	51	47169	261	76	48808	262
2	48718	404	27	34423	247	52	35545	273	77	35044	240
3	62102	451	28	26778	252	53	40807	241	78	8386	236
4	3378	399	29	50599	335	54	14747	250	79	43700	241
5	48380	304	30	54392	292	55	38823	273	80	22391	229
6	57207	298	31	35985	237	56	30988	259	81	38485	297
7	18774	375	32	52915	245	57	60012	239	82	50525	260
8	50033	301	33	32555	260	58	26405	252	83	20515	200
9	51839	271	34	23977	236	59	43391	261	84	35976	208
10	17503	267	35	11399	270	60	62070	299	85	55200	242
11	6208	285	36	59721	251	61	56955	249	86	2785	268
12	43415	262	37	60198	267	62	10047	241	87	59916	216
13	41643	282	38	10456	250	63	38623	250	88	23276	195
14	11643	310	39	8711	335	64	44674	237	89	53101	235
15	43857	307	40	61785	292	65	2745	234	90	12188	213
16	35197	266	41	39251	264	66	5551	249	91	53779	239
17	14207	269	42	29704	289	67	26578	243	92	46000	270
18	50524	266	43	40935	233	68	59994	252	93	8296	236
19	59917	265	44	58073	339	69	29044	245	94	33265	289
20	34091	245	45	21811	345	70	18367	254	95	23560	269
21	58127	287	46	21791	246	71	14704	261	96	41115	255
22	57998	262	47	29161	252	72	7710	248	97	58766	285
23	52763	278	48	50854	261	73	22871	247	98	58891	253
24	45395	270	49	6258	253	74	2266	257	99	31232	207
25	15517	298	50	53013	282	75	51893	269	100	56437	255

Table 6: Number of features used in the prediction requests: SVM model serving