May 12, 2017

ZIYAD MUHAMMED MOHIYUDHEEN

TU Berlin

Matriculation No: 387646

Serving Large Machine Learning Models on Apache Flink

IT4BI Master Thesis

Table of Contents

[1. Introduction 3](#_Toc482985536)

[2. Background and related work 4](#_Toc482985537)

[Machine learning 4](#_Toc482985538)

[Model Serving 4](#_Toc482985539)

[Model Serving Frameworks 5](#_Toc482985540)

[1. Velox 5](#_Toc482985541)

[Apache Flink 6](#_Toc482985542)

[Queryable State in Flink 6](#_Toc482985543)

[State in Flink 6](#_Toc482985544)

[Queryable State 7](#_Toc482985545)

[RocksDB state backend 7](#_Toc482985546)

[Parameter Servers 7](#_Toc482985547)

[Large Machine Learning Models 7](#_Toc482985548)

[Alternating Least Squares (ALS) 7](#_Toc482985549)

[3. Design and Implementation 8](#_Toc482985550)

[3.1 Using Queryable State 8](#_Toc482985551)

[4. Experiments and Results 9](#_Toc482985552)

[5. Conclusions 9](#_Toc482985553)

[6. Future Work 9](#_Toc482985554)

[7. References 9](#_Toc482985555)

# Introduction

# Background and related work

## Machine learning

Machine learning is the field of computer science which gives computers the ability to learn and predict without being explicitly programmed [1]. Machine learning techniques are employed in various applications in today’s digital world, which includes recommendations on online shops, weather predictions, filtering spam emails, serving ads on the web, detecting credit card fraud etc. to name just a few. 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 the whole data sets, at affordable cost and time. This has led to more feature rich models which perform better.

## Model Serving

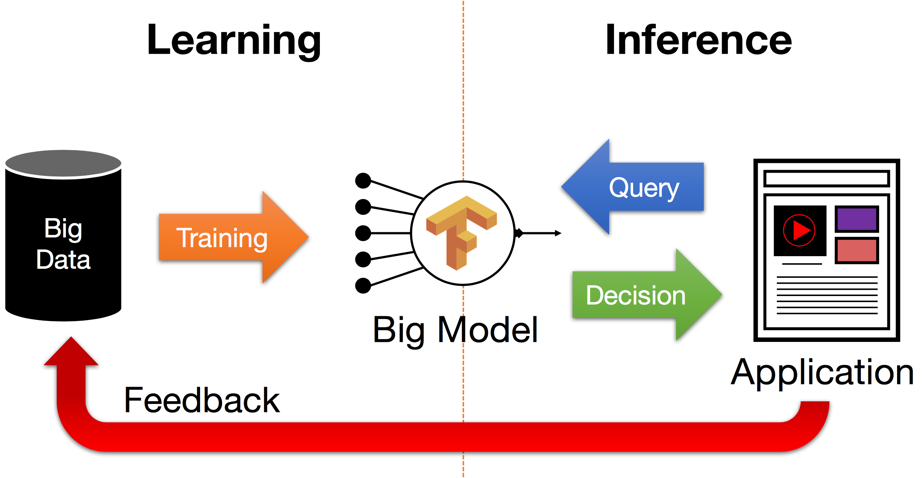


Figure 2. Model Serving of Large machine learning models [7]

As the data and features increases, the size of the machine learning model also increases. One of the critical components in machine learning applications in big data systems is the deployment of these large models, serving different applications which uses this model for predictions. The ability to update the models with the newly observed data after the initial training is also important.

Often big data frameworks are used only for the training of the 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, and the research community is now trying to address this issue to have end-to-end machine learning systems within the big data frameworks, which can perform training and updating of the models and serve various external applications.

The two approaches towards model serving are Eager and Lazy serving. In Eager, we compute and materialize all the predictions and access them in query time. Whereas in Lazy serving, prediction computation happens in real-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.

The lazy serving model 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 is 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 (< 10 ms) and high throughput.

## Model Serving Frameworks

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

***TODO***: LASER, Velox / Clipper, TensorFlow Serving, Oryx, Prediction I/O.

### Velox

Velox [5] try to address the deployment of large machine learning models in Berkeley Data Analytics Stack (BDAS). 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, and the trained model is then deployed using Velox.

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.

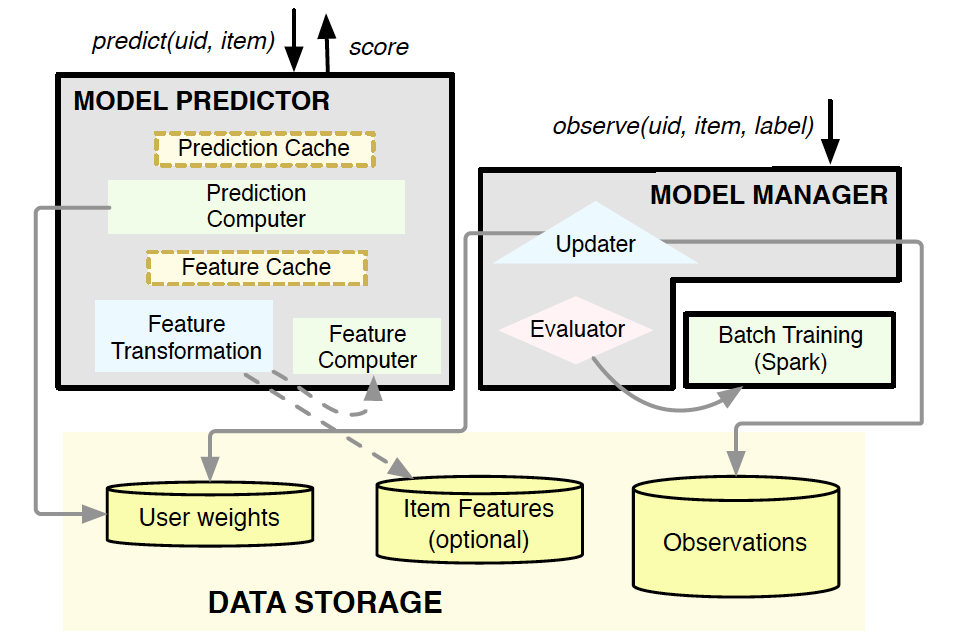


Fig.1: Velox Architecture [5]

In the prototype, Velox supports matrix factorization models which uses latent factor vectors. An example is a song prediction recommender model. The user and song latent factors of the model are deployed on Tachyon, the default storage system used in Velox, and 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 models, 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 remote data access. 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. And in order to overcome the feedback loops, Velox rely on form of a bandit algorithm, which assign an uncertainty score to each item on top of its predicted score.

## Apache Flink

Apache Flink [2] is a unified batch and stream processing framework for big data, which is often referred to as the 4th generation of big data frameworks, offering real time stream processing as well as batch processing on top of its streaming dataflow engine [3]. Its dedicated support for iterative computations including bulk and delta iterations [3] makes it ideal for many scalable machine learning applications. The machine learning library of Flink, called FlinkML [4] 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 [4].

Flink community is also actively thinking about having a serving layer on Flink, called Flink-MS [6] 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 paper, Flink-MS is only in the initial discussion phase.

## Queryable State in Flink

In this section we explore queryable state [8] feature of Flink, and how it is suitable for serving the machine learning models on Flink.

### State in Flink

Apache 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. 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 is used, where as in Raw state, operators keep their own data structures.

### 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, avoiding certain bottlenecks. This is relatively a new feature in Flink (released in version 1.2.0 in February 2017), which also includes an API to query the state from external applications. At the time of writing this paper, 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.

### RocksDB state backend

RocksDB [9] is a persistent key value store developed and maintained by Facebook Database Engineering Team, where keys and values are arbitrary byte streams. RcoksDB provides extremely low latencies 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, 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.

Queryable state along with RocksDB state backend forms an ideal candidate for building a serving layer for large machine learning models on top of Flink, which we will explore further in the section 3.

## Parameter Servers

## Large Machine Learning Models

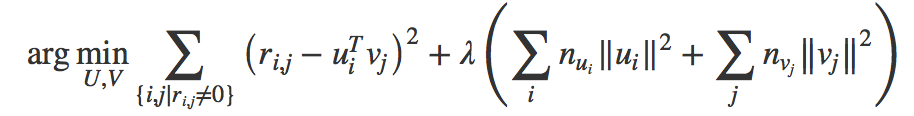
In this section, we explore some of the machine learning algorithms which produces large models.

### Alternating Least Squares (ALS)

ALS is 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≈UTV.

The number of features for the user and item matrix is given as a parameter to ALS, and is called latent factors. The ith column of U is represented as ui and the ith column of V is represented as vi. The matrix R is known as the ratings matrix with (R)i,j=ri,j

To find the user and item matrix, the below problem is solved [10]



where λ is called the regularization factor which is used to avoid over fitting. nui is the number of items the user i has rated and nvj 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.

The machine learning library of Flink provides implementation of ALS, where we can specify the number of latent factors, regularization factor and the number of iterations to perform to produce the factorization matrices U and V. For our experiments on serving large machine learning models, ALS on Flink is a good choice as we can produce large models which vary in number of features also in number of rows per matrix.

# Design and Implementation

In this section we discuss the implementation of model serving on Flink for the selected machine learning models.

## 3.1 Using Queryable State

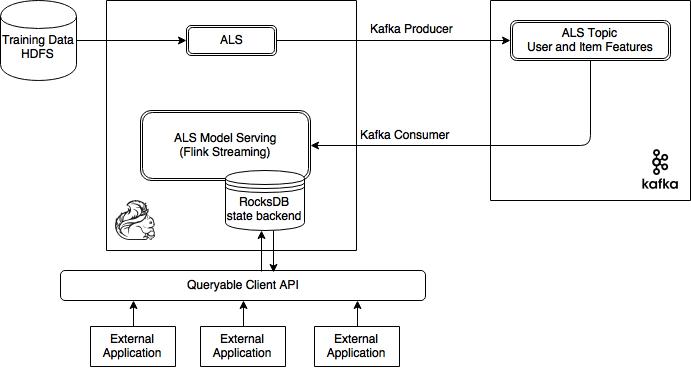


Fig.3 ALS Model Serving using Queryable state

Figure 3 shows the architecture of ALS model serving using queryable state in Flink. The first step is to obtain the matrix factorization model. Here we train the data on HDFS using Flink ALS batch job. Notice that this architecture can serve the model which is trained potentially in other platforms as well. The trained ALS model (if trained using other platforms, the model will be loaded from HDFS) is loaded into a Kafka topic, which will act as the continuous stream source for the ALS model serving Flink job. We are using Kafka cluster only to make a continuous stream source, as the queryable state life cycle is bound to the life cycle of the job in the current version of Flink. In future versions, this life cycle dependency is expected to be withdrawn, and in that case we no longer will need a Kafka source in order to serve an already trained model. We can directly load the model from a storage layer such as HDFS to the queryable state.

The second and major component in this architecture is the Flink streaming job with queryable state enabled. This job accepts the ALS trained model (from Kafka topic), and creates a keyed stream which will be stored in the RocksDB state backend as a Value state (with user or item ID as the key and the corresponding feature vector as the value). Based on the number of user and item records, we can create required number of partitions for the Kafka topic, and it will reflect in the Flink parallel operator instances and associated RocksDB instances created. Thus we can effectively distribute the user and item features across the Flink cluster.

Flink queryable client will identify the correct RocksDB instance for the queried key and state value for that key, which will be the feature vector for the queried user or item ID will be returned. Any external application which uses the queryable client remain agnostic to the RocksDB instance being queried, as Flink manages it all internally. External applications need to specify only the Job ID of the Flink streaming job and the key to be queried.

# Experiments and Results

# Conclusions

# Future Work

# References

1. <https://en.wikipedia.org/wiki/Machine_learning>
2. Apache Flink, Accessed on 28/12/2016, <https://flink.apache.org>
3. P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, Apache flink: Stream and batch processing in a single engine, Data Engineering, p. 28, 2015.
4. Apache FlinkML, Accessed on 28/12/2016, <https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/libs/ml>
5. D. Crankshaw, P. Bailis, J. E. Gonzalez, H. Li, Z. Zhang, M. J. Franklin, A. Ghodsi, and M. I. Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. In Conference on Innovative Data Systems Research (CIDR), 2015.
6. <https://docs.google.com/document/d/1CjWL9aLxPrKytKxUF5c3ohs0ickp0fdEXPsPYPEywsE/edit#heading=h.j2r3xzajbsl>
7. <https://ucbrise.github.io/cs294-rise-fa16/assets/slides/prediction-serving-systems-cs294-RISE_seminar.pdf>
8. <https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/stream/queryable_state.html>
9. <http://rocksdb.org/docs/getting-started.html>
10. <https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/libs/ml/als.html>