Co-scheduling Feature Updates and Queries for Feature Stores

Woosuk Kwon, Debbie Liang, Jimmy Xu

December 15, 2021

Abstract

Feature stores are fast emerging as a new class of Machine Learning system that maintains intermediate statistics of live data streams used for model training and inference. Our work explores the impact of *lazy evaluation* on the three most important aspects of feature stores (i.e., staleness, latency, and costs) and builds an *SLO-aware featurization scheduler* that reduces the staleness of the queried features by co-scheduling feature updates and query responses. When evaluated on an online recommendation workload, our system shows up to 14% accuracy improvement compared to the baseline feature store system while meeting the SLOs for 99.7% of the queries.

1 Introduction

In real-world machine learning applications, it is a common practice to pre-process data and get features to reduce latency and improve accuracy. Features are raw and derived data and can have different representations and encodings. The process of converting the data into features is called featurization. Machine learning models need to featurize data from the data source to do training or prediction. For example, in the scenario of credit card fraud detection, the application uses credit cards' activity patterns to identify anomalies, while the user interface of a bank can provide only raw data such as credit card number, individual transaction record time, and transaction amount. For the machine learning model to detect fraud, some pre-computation needs to happen to produce the probability of each transaction occurrence, so that the downstream model can infer if the transaction is a fraud.

Featurization usually takes a significant amount of computing power and time. It is desirable for online models to respond in a timely manner to achieve its goal (e.g. stop fraudulent transactions in time). Thus, a feature store that can pre-process raw data and respond to queries from models is ideal to hide the latency associated with featurization.

Feature stores usually store and maintain a feature table that maps from a feature key (e.g. a user ID) to featurized data (e.g. the transaction history of a user). Ideally, the table should be updated upon receiving new raw data; however, this will require a massive amount of computation and may result in higher latency. Thus, a feature store needs to decide on the frequency of update and balance the tradeoffs between latency and staleness of the features.

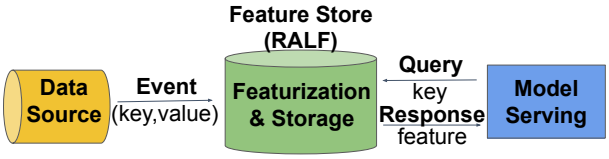

Figure 1: Overview of a Machine Learning pipeline with a Feature store: A data update creates an event with a key and a value. The feature store performs featurization and stores the feature. The Downstream model can issue a query with the desired key and the feature store will respond with the feature value associated with that key.

Among the many exploration of feature stores, RALF [23] is designed for streaming data, and it explicitly leverages downstream feedback to reduce costs with minimal downstream accuracy degradation. Currently, RALF only computes updates eagerly. As Figure 2 shows, the latency and staleness of eager computation increase exponentially when the rate of incoming data, or update rate, is around or above 1000 per second. Our project adds the support for lazy computation to RALF to mitigate this issue while exploring the tradeoffs associated with lazy computation. We use the metrics defined in the following section to evaluate the effectiveness of our implementation. Another

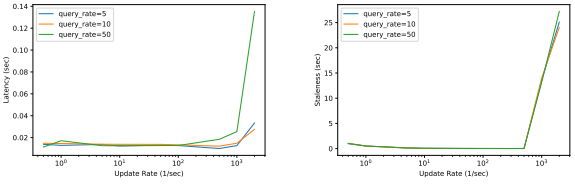

Figure 2: Latency and staleness of eager evaluation

issue is that RALF does not consider query service-level objectives (SLOs) for scheduling queries and features updates. As a result, they lose the opportunities to return

more up-to-date features computed from the freshest data updates. In our project, we explored the query SLOs between 10ms to 50 ms to see how different scheduling policies and latency requirements can affect the accuracy of the downstream model. With the co-scheduling algorithm, features are computed when queried to meet some specified deadline while increasing the accuracy.

The rest of this paper is structured as follows. In section 2, we discuss the success metrics that our system considers. Then, we describe our exploration of metrics tradeoff with synthetic data in section 3 and our scheduler implementation in section 4. In section 5, we present the experiment results of our scheduler implementation. We then have a little discussion of the experiment results in section 7, compare our scheduler to related work in section 6, and conclude in section 8.

2 Success Metrics

When training machine learning models with a feature store, developers care the most about three metrics.

Staleness Different workloads might have different definitions of staleness but in general, the more stale the features are, the less fresh they are, and the lower accuracy the model output can have. Given the fact that the environment might change a lot when time goes by, fresher features tend to reflect the real environment more accurately. Downstream model accuracy is a clean and effective way to measure the impact of feature store staleness. In our work, we will explore different ways to reduce staleness of our features and increase accuracy for two different workloads.

Latency Latency is defined as the time difference between the time a query is issued and the time a response is received. To achieve higher pipeline efficiency and also a better user experience, we pursue a lower latency when possible. In this project, we either measure the latency directly or give an explicit SLO constraint to the feature store when issuing a query.

Computation cost Featurization in feature stores can be very expensive. For example, in our embedding workload, a featurization can take a few milliseconds and there can be a few hundred data updates per second. In real world, this can be translated to AWS lambda [1] rental costs. The more computation that happens, the more developers need to pay for a serverless compute service. We aim to reduce the number of times calling the featurization methods.

In the following sections, we explored the trade-offs between these three metrics.

3 Exploring Trade Offs in Lazy Evaluation

We implemented lazy evaluation and a load shedding policy. We then created a synthetic workload to understand the behavior of the current system as well as exploring various trade offs between lazy evaluation and eager evaluation.

3.1 Lazy Evaluation Implementation

In the current implementation of RALF, it always does featurization upon receiving new data (i.e. eager evaluation). We implemented lazy evaluation with which the system won't featurize the updated raw data unless the client queries for it. Upon receiving a query, if the current table has the feature, the system will return the stale feature immediately and schedule a featurization. If the current table does not have the corresponding feature but the key exists in an upstream table, it will wait for the featurization to complete. Otherwise, a key error will be returned indicating the key does not exist in the system.

As illustrated in Figure 3, eager evaluation will perform featurization for updates of (key_1,value_1), (key_1,value_2), and (key_1,value_3), respectively. Lazy evaluation will not do anything until the first query with key_1 comes in. Lazy evaluation doesn't have a feature value for key_1, so it checks its upstream table and see that an update with (key_1, value_3) is the latest update with key_1. Lazy evaluation performs a featurization and returns a feature associated with value_3. Eager evaluation returns the feature associated with value_3 that it has computed. When update (key_1, value_4) arrives, eager evaluation will perform another featurization while lazy evaluation does nothing. When query with key_1 comes in again, eager evaluation will return the feature associated with value_4 that it has computed. Lazy evaluation sees that it has a feature for key_1 that was computed from value_3, so it will return value_3 to the query first, and schedule a featurization so that it can respond with the feature associated with value_4 when the next query comes in.

3.2 Load Shedding

A disadvantage of laziness is when the query rate is high, it may schedule unnecessary updates. To avoid redundant computations, our system provides an option to group N queries and check if there is duplicate update requests for the same key. The result of this policy is illustrated in the subsection below.

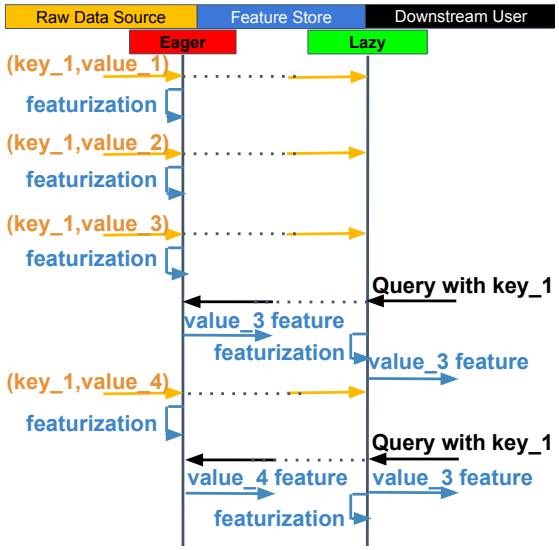

Figure 3: Lazy evaluation workflow: the lazy computation only does featurization when receiving a query.

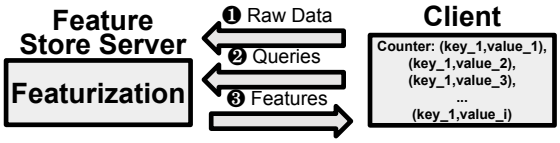

Figure 4: Synthetic Workflow with a server and a client

3.3 Synthetic Workload

As shown in Figure 4, the server simulates the featurization pipeline. The client continuously generates raw data and sends HTTP requests to the server. The server has a source operator that receives streaming data through a Redis [17] data structure store. The server then performs featurization with a featurization operator. The client also simulates a downstream application that gets desired features by querying the table in the featurization operator. To avoid the impact of network connections, we assume that the clients, the inference server, and the feature store are running on the same local machine with varying parameters.

The implementation details are as follows. The client has `num_keys` of keys. It will count from 0 to `update_up_to` and assign the number (i.e. raw data) to key `num % num_keys`. Each data will be generated every `1/update_rate` seconds and sent to the source operator on the server. The featurization operator performs an identity operation that is set to take `processing_time` seconds, simulating an expensive featurization operation. The client sends a query every `1/query_rate` seconds.

3.4 Experiments

We conducted a number of experiments on the synthetic workload to explore: All experiments are done with the following parameters:

(1) There only exists one key.
(2) The featurization time is set to be 0.01 seconds.
(3) Unless specified, load shedding policies are not enabled.
(4) The query requests and update requests are in the same queue and are processed using FIFO policy.
(5) There are 4 worker threads to process queries and updates.
(6) The client continuously queries the server for 5 seconds, three times (15 seconds in total). The latency and staleness is the average of the last two runs, which excludes potential initialization time.

Trade offs We expect a higher staleness for lazy evaluation because when there is an existing value for a certain key being queried, the value will be returned immediately. Since featurization only happens when there is a query to the key, if the last time this key got queried was a long time ago, the value returned would not be very fresh (i.e., the staleness of the data is very high).

We expect a comparable latency for lazy evaluation if the update rate and query rate are within a normal range, because for a key that shows up more than once, both lazy and eager evaluation will return an old value that they computed from the last query or data update, respectively. However, if updates and queries come in from the same data stream and each query result needs to wait for the corresponding update (e.g., real-time click-recommendation on commercial websites), lazy evaluation may have a lower latency because it doesn't need to wait for the featurization to complete.

Since featurization is the most expensive process, we expect lazy evaluation to have a higher computation cost when the query rate is high because it does featurization whenever a query comes in. For the same reason, we expect eager evaluation to have a higher computation cost when the update rate is high because it does featurization for every data update.

To confirm our hypothesis about metrics changes, we varied the update rate and query rate for both lazy evaluation and eager evaluation, and measured the latency, staleness, and computation cost.

Load shedding In both eager and lazy evaluation, our load shedding policy can group N update requests for the same key so that only one update is performed, as opposed to N. This decreases the number of updates and increases the limit of concurrent updates and queries.

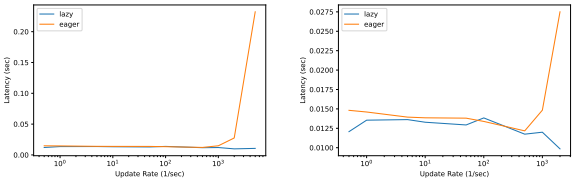

Figure 5: Latency of eager and lazy evaluation with fixed query rate of 10.
Left: Zoomed out. Right: Zoomed in.

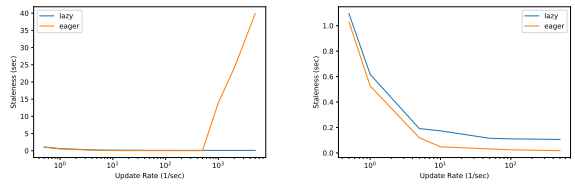

Figure 6: Staleness of eager and lazy evaluation with fixed query rate of 10.
Left: Zoomed out. Right: Zoomed in.

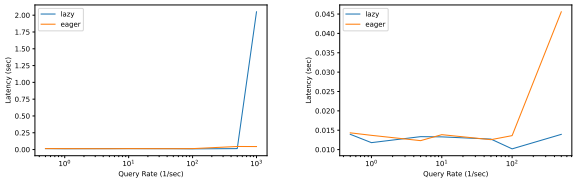

Figure 7: Latency of eager and lazy evaluation with fixed update rate of 10.
Left: Zoomed out. Right: Zoomed in.

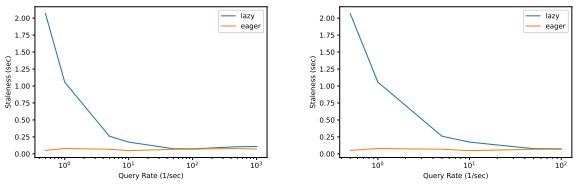

Figure 8: Staleness of eager and lazy evaluation with fixed update rate of 10.
Left: Zoomed out. Right: Zoomed in.

3.5 Observations

Varying the update rate Figure 5 and Figure 6 shows that when the query rate is fixed, the latency and staleness for the feature store with lazy updates and eager updates are around the same when the update rate is below 1000 times per second. For a higher update rate, the latency of the feature store with eager computation grows exponentially.

Varying the query rate Figure 7 and Figure 8 shows that when update rate goes high, the eager evaluation will encounter a congestion problem because the limited computing resources cannot process all the updates. This leads to a rise in the latency and staleness of records. Lazy evaluation mitigates this problem by postponing processing new updates until receiving queries. For the synthetic data, only the latest value associated with the same key matters so as long as the query rate stays the same, the lazy evaluation only performs certain amount of computation. In addition, since featurization only happens when receiving a query in lazy evaluation, if the query rate is too low (< 10 queries / second), the staleness can be high, as shown in Figure 8. In this case, too many updates were discarded because of the infrequent queries. As query rate grows, the staleness of data for a feature store with lazy evaluation and eager evaluation converges.

Computation costs With similar reasons as above, more computation leads to more cost. When update rate goes high, the cost for a feature store with eager evaluation goes high. When query rate goes high, the cost for a feature store with lazy evaluation goes high. The plateaus in Figure 9 indicates the system cannot process more updates concurrently.

Load shedding As expected, our load shedding policy can decreases the number of redundant featurization and prevent the computation cost from growing too fast. The results for lazy computation with and without load shedding policy are shown in Figure 10.

The current implementation of RALF treats update requests in the same way as query requests. It only has one queue to process both events, which may result in unnecessary waiting time for time-sensitive queries. Prioritizing queries is discussed in later sections.

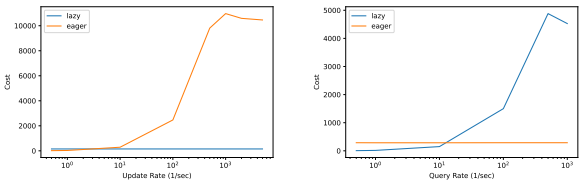

Figure 9: Computation costs of eager and lazy evaluation with fixed query rate of 10 (left) and with fixed update rate of 10 (right).

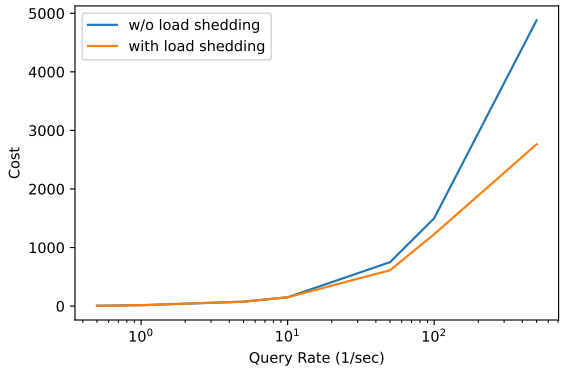

Figure 10: Cost between lazy evaluation with and without load shedding with fixed update rate of 10

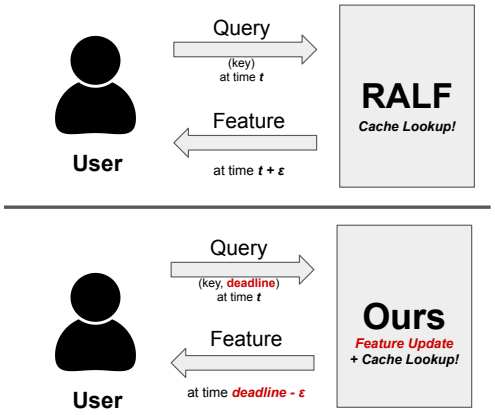

Figure 11: The high-level comparison between RALF and our system. While RALF is designed to immediately respond to queries by a simple cache lookup, our system provides a way for users to specify their query deadlines and considers the deadline information in scheduling.

4 SLO-aware Featurization Scheduler

In this section, we design and implement an SLO-aware featurization scheduler based on the explored trade offs in Section 3. Motivated by the fact that the ultimate goal of real-time ML serving systems is to provide high-quality predictions while meeting SLOs, we argue that the goal of feature store should be to provide high-quality features within the query deadlines. We show that by considering SLOs our scheduler can provide the most up-to-date features to queries and thus significantly enhance the downstream model accuracy.

Figure 11 illustrates the key difference between RALF [23], the existing feature store system, and our system. In RALF, the task of responding to queries and the task of updating features are done *asynchronously*. Specifically, RALF treats a feature query as a lookup to its cache store which is continuously updated by the incoming stream of events. As such, because individual queries are responded without any waiting time by RALF, their latencies are extremely low (i.e., < 5 ms) and stable. Nevertheless, RALF loses opportunities to return more up-to-date features by co-scheduling query responses and feature updates. For example, when a query and an event to the same feature arrive simultaneously, RALF cannot reflect the latest event in the query response. For many real-time applications sensitive to feature staleness, this can lead to significant accuracy loss. Our scheduler addresses this limitation by accepting the deadline of each query and using it to intelligently schedule the featurization pipelines.

4.1 System Overview

The fundamental idea of our scheduler is to prioritize the updates of the features being queried over those not being queried. That is, our system processes the latest events of

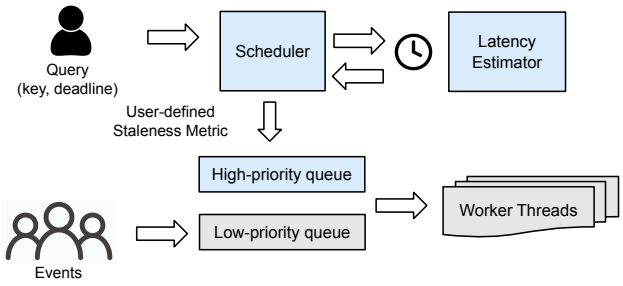

Figure 12: Overview of our system. The blue boxes represent the system components we added on top of RALF. Our system accepts deadline-specified queries, and co-schedules query responses and feature updates, using its latency estimator and a staleness metric defined by the user.

the features upon requests and returns the up-to-date features to the queries within their deadlines. The intuition behind this is that the system can process the deferred updates (i.e., updates for unqueried features) when its resources are underutilized, as the distribution of the query arrival time is naturally skewed.

However, when the query rate is high and the system has a fixed processing capacity, the scheduler cannot process the updates of all the queried features while meeting the SLOs. Thus, among the features being queried, the scheduler should select which features to update before the deadlines and which features to not. To this end, our scheduler takes a *staleness-based policy* that gives higher priority to the processing of staler features over the updates for fresher features. That is, when the query rate is beyond the system capacity, our scheduler selects the queries to the stalest features and returns the features after updating them. For

the queries that are not selected by the scheduler, features cached in the system are simply returned without updates, as in the current version of RALF.

Figure 12 illustrates our overall system architecture. Our system augments the RALF's query API so that users can specify the deadlines of their queries. Our scheduler then makes use of the SLO information and the estimated latency of feature updates to calculate the maximum number of queries whose feature updates can be processed before the deadlines. When the number of pending queries exceeds this number, queries to the stalest features are selected based on a *user-defined staleness metric*. Updated features are served for the selected queries, while cached (possibly stale) features are return for those not selected. When the query rate becomes low and the system resources are underutilized, the deferred updates for queried/unqueried features are processed.

4.2 Scheduling Policy

Formally, the scheduling granularity of our system is an individual event, a collection of the tasks to update a certain feature with new data. While the ways to update features may vary, we only focus on such a case that an ML algorithm is involved in the featurization pipeline, and thus each event takes non-negligible time (e.g., > 1 ms) to be processed. For example, in a recommender system, user features are often computed by a computationally heavy ML model and used for making predictions. And the user features are continuously recomputed as more user data (e.g., the user's page views) are collected. In this example, updating a user feature by running the ML model with new data is regarded as a (processing of) event. As such events stream into the system, the system scheduler should decide which events to process first and which ones to process later. The decision of the scheduler can largely affect the downstream model accuracy, as providing stale features to downstream applications may lead to bad predictions (e.g., recommending items irrelevant to the user's latest page views).

Two-level Priority. The core of our scheduling policy is assigning two-level priorities to events. Specifically, the higher priority is given to the events whose target features are being queried and the lower priority is given to those whose target features are not being queried. The priority is strict; events with the lower priority are never processed as long as there exists any higher-priority event. Also, the scheduling algorithm is non-preemptive; because each event usually takes a few milliseconds, the system simply awaits when a low-priority event blocks a high-priority one.

Staleness-based Prioritization. When the query rate is high, the scheduler cannot process the events of all queried features while meeting the SLOs. The selection of events to process before deadlines is based on the degree of staleness of the queried features. Specifically, the scheduler prioritizes processing events that recompute stale features over the those for fresh features. Our intuition is that, if the computation costs are the same, updating a stale feature is more likely to improve the overall downstream accuracy than updating a fresh feature. The system relies on users to define their staleness metrics. This allows users to leverage their domain knowledge to further enhance scheduling.

Note that the priority of the unselected events is deescalated to the low priority. This ensures an important invariant that at every moment the number of the high-priority events in the system do not exceed the maximum number of events that the system can process within the deadlines. We find this invariant extremely helpful in bounding the query latency and efficiently managing the queue of the high-priority events.

Fairness and Starvation. Arguably, our scheduling policy maintains *fairness in terms of feature staleness* in that it essentially aims to equalize the staleness of the stored features. Moreover, we can guarantee that our scheduling policy has no starvation problem. One might point out that the events for unqueried features will never get processed if the query rate is continuously high and the features are never queried. However, this is actually desirable, as the system is saving the computation cost for updating the features that will not be queried afterwards. When such a feature is queried, the scheduler will likely find it much staler than others and prioritize the processing of its events.

4.3 Scheduling Mechanism

As in Figure 12, we use multi-level queues for the two-level priority scheduling. The both queues use first-in-first-out (FIFO) processing policy. The detailed scheduling mechanism is as follows.

When a query arrives, the system first computes the staleness of the queried feature using the user-defined staleness metric function. The function takes the cached feature and its latest event as the inputs, and express the feature staleness in a non-negative real value. The 0 staleness indicates that the feature is in its freshest state while larger values mean that the feature is staler. Here, queries to the features that are not yet created by pending events get the maximum level of staleness.

Usually, the system has unprocessed events in the low-priority queue for the queried feature. The scheduler predicts whether these events can be processed before the query deadline. The prediction is made by considering the number of events in the high-priority queue that should be processed for preceding queries and the processing time of an event estimated by the *latency estimator*. The latency es-

timator periodically measures the processing time of the individual events and provides the exponential moving average of the processing time to the scheduler.

The events for the queried feature are simply put into the high-priority queue if they can be processed before the query deadline. If not, the scheduler tries to reduce the processing time of the query by removing some of the unprocessed events in the high-priority queue. Specifically, if the high-priority queue contains an event to update a feature less stale than the queried one, the scheduler removes that event from the queue. This process is repeated until the estimated processing time of the query becomes before the query deadline or no event can be further removed from the queue by the feature staleness comparison.

Lastly, the updated feature is stored in the cache and served to the querying user. Importantly, we design our system to *asynchronously* perform the task of recomputing features and the task of responding to queries (just like RALF). Regardless of the scheduling state, the system retrieves queried features from its cache at the moment right before the query deadlines. For example, if a query latency budget is 100 ms, the system will simply return the feature stored at the moment when 99 ms has been elapsed from the query arrival. Although such a trick can lead to higher average latency because of the unnecessary delay in query responses, we find it necessary for simplifying the system design and strictly bounding the latency even when the latency estimation is inaccurate.

4.4 Implementation

Our system is implemented on top of RALF, which is in turn built on top of Ray Serve [12, 16], a scalable model serving library. Our scheduler is implemented in ~300 lines of Python code. We also provide default staleness metric functions that use either the elapsed time from when the feature is lastly computed or simply a random value.

5 Evaluation

We showcase the effectiveness of our system on an online recommendation workload. Specifically, we evaluate the system on a next item prediction task, a core recommendation problem whose goal is to predict the next item that the user is most likely to interact with given the sequence of the user's previous items. Obviously, in order to achieve high accuracy on the next item prediction task, it is important to consider the users' latest interaction with items to infer their current interest. In this section, we show that our system enables this by co-scheduling query responses and feature updates whereas RALF produces stale features and results in significantly low accuracy.

5.1 End-to-End Workflow

Figure 13 illustrates the end-to-end recommendation system pipeline developed for our experiments. Imagine that you are shopping on an online store such as Amazon. When you click into an item page, the site will also show other items you may be interested in. For example, if you are browsing smartphones, the site will probably display ones from different vendors or some accessories for the phones. In personalized recommendation systems, the recommended items are not only relevant to the one you are currently viewing but also have connections with your recent browsing history, which better represents your current interest. If the recommendation is successful, you will click through one of the items. The end goal of the system is to increase the likelihood that you click one of the recommended items at every browsing step.

The process of making recommendations is collaboration of two separate systems, an inference server and a feature store (i.e., either our system or RALF). The inference server is responsible for accepting user inputs, making final predictions, and sending them back to the users. The predictions are made using the features provided by the feature store, which are in this case the *user embeddings* produced by an ML model called XLNet4Rec [20].

The system workflow can be described as follows. ❶ Whenever a user clicks an item, ❷ the inference server queries the user embedding to the feature store. The key of the query is the user ID, and the deadline of the query is given by the inference server. Here, at the same time the user embedding is queried, ❷ the inference server also creates an event for updating the user embedding with the new item ID. That is, a user click works as both a query and an event, since a click is a request to a new recommendation but is also used to recompute the user embedding. ❸ After the feature store returns the queried feature, ❹ the inference server makes predictions via similarity search between the user embedding and the item embeddings. ❺ Finally, the inference server serves the list of recommended items to the user.

5.2 Workload Generation

Base Benchmark. Our workload is generated from the REES46 E-commerce dataset [1], which contains real customer behavior data from a large online store. The dataset includes the log of user-item interactions, such as page view, add-to-cart, and purchase, with the timestamps and item metadata. For simplicity, we do not distinguish the types of interactions and regard them equally as a click. Additionally, we ignore the item metadata and only use the tuples of (user_id, item_id, timestamp) in training and

[1] https://rees46.com/en/datasets/

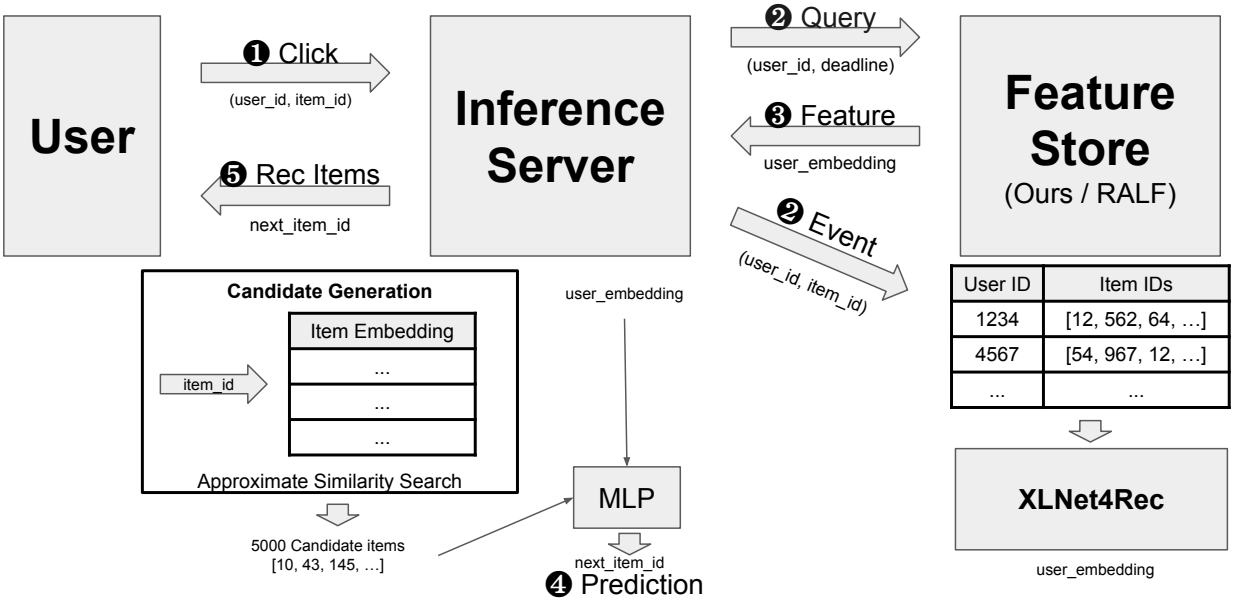

Figure 13: End-to-end recommendation serving pipeline used for our experiments.

evaluating our recommendation model. Lastly, out of the seven-month data in the dataset, we only use the events from the month of October 2019 for our experiments, as in [10].

Recommender Model. Following the recent findings that Transformer models [21, 2, 20] achieve state-of-the-art performance in the next item prediction tasks, we use XLNet4Rec [20] as our recommender model. Similar to BERT4Rec [21], XLNet4Rec consists of an item embedding table and 2 Transformer blocks, each of which has 4 attention heads and hidden dimension of 448. The model takes as the input a sequence of the item IDs that a user has interacted with, and encodes it to a feature that densely represents the user interests, which we call a *user embedding*. Then the items whose embeddings are most similar to the user embedding are recommended to the user. The model is incrementally trained on the data from October 1st to October 30th, and is tested on the October 31st data.

In the test data, we find that the peak number of clicks per second is only 25, which is insufficient to show the impact of our scheduler. To evaluate our system on a more intensive workload, we synthesize a click pattern by adjusting the initial timestamps when users starts their clicks. This way, we get a click pattern where the time interval between the user's subsequent clicks is preserved while more clicks arrive simultaneously to the system. From the synthesized workload, we extract the peak 10 minutes data, which has at most 250 clicks per second.

5.3 Experimental Setup

All of our experiments in this section are conducted on an AWS c5d.9xlarge instance [2], which has 48 virtual CPUs. Note that, following the common practice in recommendation systems [10], all the computations including the inference of XLNet4Rec are processed on CPUs, not GPUs.

The code of XLNet4Rec is brought from the NVIDIA Transformers4Rec repository [13]. We train the model with a standard training recipe. At inference time, the model is executed on PyTorch [15] which is powered by the Intel MKL-DNN library [11].

The inference server is implemented using Ray Serve. For the approximate similarity search, we use the CPU version of the Faiss library [6]. In querying to the feature store, the inference server sends HTTP requests. For sending new data to the feature store, the inference server uses Redis Stream [17]. To avoid the impact of network connections, we assume that the clients, the inference server, and the feature store are running on the same machine.

5.4 Downstream Accuracy

Figure 14 shows the downstream model accuracies of RALF and our system. We use top-20 hit rate as our accuracy metric, which is the probability that the true next item belongs to the top-20 recommendations. In the optimal case where the recommendation is always made on the most up-to-date features, the accuracy is 42%. This is the

[2] https://aws.amazon.com/ec2/instance-types/c5/

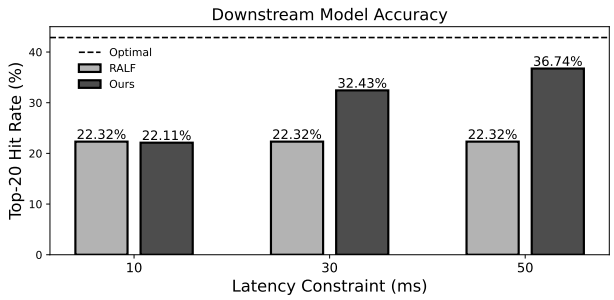

Figure 14: Recommendation accuracy of RALF and our system with varying latency constraints.

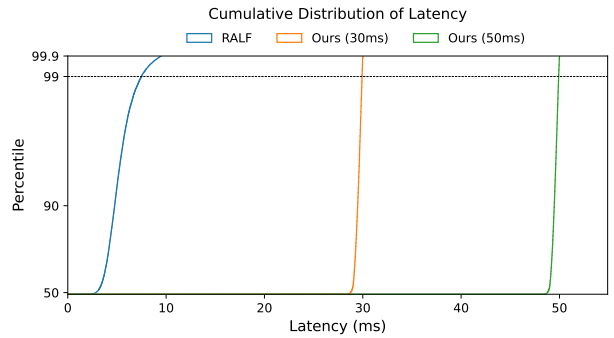

Figure 15: Cumulative distribution of query latencies of RALF and our system, with two different latency constraints (i.e., 30 ms and 50 ms). Because the latency of RALF does not change by the latency constraint, it is drawn once. The y axis is scaled to highlight the SLO violation rates.

upper bound of the accuracy that we can obtain from our XLNet4Rec.

Our system shows up to **14%** improvement in accuracy compared to RALF when the latency constraint is 50 ms. The huge accuracy gain mostly comes from our co-scheduling policy that enables updating queried features before returning them. Particularly in our next item prediction task, this difference is critical in accuracy; a query and an event (from the same click) always arrive simultaneously, and the users' latest click information is a key ingredient in predicting their current interests. Because RALF cannot immediately reflect this information in the queried features, it suffers from significantly low accuracy. On the other hand, when a sufficient latency budget (e.g., $\geq$ 30 ms) is given, our system recovers most of the accuracy loss by intelligently prioritizing the updates for queried features.

5.5 Query Latency

Figure 15 shows the cumulative distribution of query latencies of RALF and our system when 30/50 ms latency constraints are given. Note that the latency (and actually the behavior) of RALF does not change according to the latency constraints, as RALF does not use the SLO information in scheduling.

As discussed in Section 4, RALF retains low and stable latency since it processes every feature query as a cache lookup. In contrast, our system shows much higher average latency as the system intentionally postpones responding to queries until their deadlines. However, our system still meets the SLO for more than **99.7%** of the queries, as shown in Figure 15. This is because the system falls back on its cached feature store when the processing time of queries accidentally exceeds the deadlines. We believe that, under such low SLO violation rates, the accuracy improvement by our scheduler far outweighs the potential advantages of the unnecessarily low latency of RALF in most cases.

6 Related Works

Recently, many ML serving systems have been proposed both in academia [4, 5, 9, 18, 24, 3, 19] and industry [22, 16, 14]. The main goal of these systems is to serve as many ML queries as possible while meeting the SLOs. Their core techniques are batching, load balancing, and auto scaling [8], which are not covered in detail in our paper.

A limitation of the existing ML serving systems is that they only handle *stateless* inference jobs, such as object detection and text classification, where no intermediate data (i.e., features) persist after the completion of the job. In essence, the systems regard an inference job as running a pure function that receives model inputs and returns a prediction. However, this is not the case in many ML serving pipelines; ML models often use pre-computed features for their inference, which should be cached and continuously updated over time by the system.

Feature store has emerged as the system layer responsible for storing the pre-computed features. Feast [7] is an open-source feature store that manages fast retrieval of features with consistency. However, Feast itself does not manage the featurization pipeline, and thus cannot exploit the opportunity to improve its accuracy/computation cost trade off by intelligently scheduling the feature updates. RALF [23] addresses this limitation by leveraging downstream feedback. It supports a fine-grained scheduling mechanisms over keys and periodically controls the feature update frequency to reduce the computation cost. Our system extends RALF to more intelligently schedule feature updates by exposing the SLO information of queries to the scheduler.

7 Discussion

Feature store is a very new concept in the system design area and researchers are currently conducting a series of exploration of its functionality, effectiveness, and tradeoffs. The three metrics that we propose are tentatively used by RALF and we regard them as the most important attributes of feature store research. In our exploration of feature store tradoffs with synthetic data, we've observed that lazy updates can cause high latency when query rate goes high and causes a congestion. On the other hand, if we can resolve the query congestion issue with some load shedding policy, lazy computation has some advantage in decreasing latency by avoiding waiting for featurization. Laziness also saves some compute power by skipping some updates when not all the updates are important for the downstream model. Additional update rules can be added to RALF by extending its load shedding method library, or adding specific data types that stores updates with different desired attributes. The synthetic data is currently generated from a counter workload, which doesn't use much CPU of the machine. In future works, we will replace it with a random dot product calculation to represent real world workload more precisely.

In Section 4, we rely on many assumptions to simplify the problem. First, we assume that our system runs on a single node, which has a fixed amount of CPUs. Thus, one of our future works would be to extend our scheduler to work on multiple nodes with heterogeneous devices (e.g., GPUs and TPUs). Second, our system does not consider batch processing of events for different features. As batching can greatly improve the system utilization in ML workloads, we can expect that introducing batching algorithms to our scheduler will further enhance the downstream accuracy. Lastly, our scheduler assumes that the system has no malicious users. Since the queries to non-existent features (i.e., those not yet created by the pending events in the low-prirority queue) get the highest priority, a malicious user can block the processing of other users' queries by continuously sending queries to new features. In our future work, we will explore a method (e.g., priority boosting) to protect the system from such an attack.

8 Conclusion

In this paper, we first explored the trade offs in designing a feature store. The main factors we considered are query latency, feature staleness, and computation cost. By comparing the eager and lazy evaluation mechanisms, we find that the optimal scheduling algorithm should adaptively use either of the mechanism, depending on the workload intensity.

Based on the explored trade offs, we designed and implemented an SLO-aware featurization scheduler on top of RALF. It allows users to specify their query deadlines, and co-schedules feature updates and query responses to reduce the staleness of the queried features. In our experiments, we show that our system improves the downstream accuracy by up to 14% while achieving the SLO violation rates less than 0.3%.

References

[1] *AWS Lambda*. https://aws.amazon.com/lambda.

[2] Qiwei Chen et al. "Behavior sequence transformer for e-commerce recommendation in alibaba". In: *Proceedings of the 1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data*. 2019, pp. 1–4.

[3] Yujeong Choi, Yunseong Kim, and Minsoo Rhu. "Lazy Batching: An SLA-aware Batching System for Cloud Machine Learning Inference". In: *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2021, pp. 493–506.

[4] Daniel Crankshaw et al. "Clipper: A low-latency online prediction serving system". In: *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 2017, pp. 613–627.

[5] Daniel Crankshaw et al. "InferLine: latency-aware provisioning and scaling for prediction serving pipelines". In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. 2020, pp. 477–491.

[6] *Faiss library repository*. https://github.com/facebookresearch/faiss.

[7] *Feast: Feature Store for Machine Learning*. https://feast.dev.

[8] Anshul Gandhi et al. "Autoscale: Dynamic, robust capacity management for multi-tier data centers". In: *ACM Transactions on Computer Systems (TOCS)* 30.4 (2012), pp. 1–26.

[9] Arpan Gujarati et al. "Serving DNNs like clockwork: Performance predictability from the bottom up". In: *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 2020, pp. 443–462.

[10] Udit Gupta et al. "The architectural implications of facebook's dnn-based personalized recommendation". In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2020, pp. 488–501.

[11] *Intel oneDNN library repository*. https://github.com/oneapi-src/oneDNN.

[12] Philipp Moritz et al. “Ray: A distributed framework for emerging {AI} applications”. In: *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 2018, pp. 561–577.

[13] *Nvidia Transformers4Rec repository*. https://github.com/NVIDIA-Merlin/Transformers4Rec.

[14] *NVIDIA Triton Inference Server*. https://github.com/triton-inference-server/server.

[15] Adam Paszke et al. “Pytorch: An imperative style, high-performance deep learning library”. In: *Advances in neural information processing systems* 32 (2019), pp. 8026–8037.

[16] *Ray Seve: Fast and simple API for scalable model serving*. https://www.ray.io/ray-serve.

[17] *Redis*. https://redis.io.

[18] Francisco Romero et al. “{INFaaS}: Automated Model-less Inference Serving”. In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 2021, pp. 397–411.

[19] Haichen Shen et al. “Nexus: a GPU cluster engine for accelerating DNN-based video analysis”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019, pp. 322–337.

[20] Gabriel de Souza Pereira Moreira et al. “Transformers4Rec: Bridging the Gap between NLP and Sequential/Session-Based Recommendation”. In: *Fifteenth ACM Conference on Recommender Systems*. 2021, pp. 143–153.

[21] Fei Sun et al. “BERT4Rec: Sequential recommendation with bidirectional encoder representations from transformer”. In: *Proceedings of the 28th ACM international conference on information and knowledge management*. 2019, pp. 1441–1450.

[22] *TensorFlow Serving*. https://github.com/tensorflow/serving.

[23] Sarah Wooders et al. *RALF: Accuracy-Aware Scheduling for Feature Store Maintenance*. URL: https://github.com/feature-store/ralf.

[24] Chengliang Zhang et al. “Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving”. In: *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 2019, pp. 1049–1062.