RALF: Accuracy-Aware Scheduling for Feature Store Maintenance

Sarah Wooders
UC Berkeley
wooders@berkeley.edu

Xiangxi Mo
UC Berkeley
xmo@berkeley.edu

Amit Narang
UC Berkeley
narang@berkeley.edu

Kevin Lin
UC Berkeley
k-lin@berkeley.edu

Ion Stoica
UC Berkeley
istoica@berkeley.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@berkeley.edu

Natacha Crooks
UC Berkeley
ncrooks@berkeley.edu

Joseph E. Gonzalez
UC Berkeley
jegonzal@berkeley.edu

ABSTRACT

Feature stores (also sometimes referred to as embedding stores) are becoming ubiquitous in model serving systems: downstream applications query these stores for auxiliary inputs at inference-time. Stored features are derived by *featurizing* rapidly changing base data sources. Featurization can be costly prohibitively expensive to trigger on every data update, particularly for features that are vector embeddings computed by a model. Yet, existing systems naively apply a one-size-fits-all policy as to when/how to update these features, and do not consider query access patterns or impacts on prediction accuracy. This paper introduces RALF, which orchestrates feature updates by leveraging *downstream error feedback* to minimize *feature store regret*, a metric for how much featurization degrades downstream accuracy. We evaluate with representative feature store workloads, anomaly detection and recommendation, using real-world datasets. We run system experiments with a 275,077 key anomaly detection workload on 800 cores to show up to a 32.7% reduction in prediction error or up to 1.6× compute cost reduction with accuracy-aware scheduling.

PVLDB Reference Format:
Sarah Wooders, Xiangxi Mo, Amit Narang, Kevin Lin, Ion Stoica, Joseph M. Hellerstein, Natacha Crooks, and Joseph E. Gonzalez. RALF: Accuracy-Aware Scheduling for Feature Store Maintenance . PVLDB, 17(3): 563-576, 2023.
doi:10.14778/3632093.3632116

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/feature-store/ralf.

1 INTRODUCTION

Most real-world applications of machine learning rely heavily on pre-computed *features* to improve model accuracy and reduce prediction latency. Features are raw and derived data that are passed as input to machine learning models to capture the context around a prediction. For example, fraud detection and content recommendation models rely on features describing merchants, users, and content to make accurate predictions. More recently, large language models increasingly depend on features of relevant context (eg. embeddings of past conversational history) to provide more grounded and personalized responses [25, 35, 36, 44].

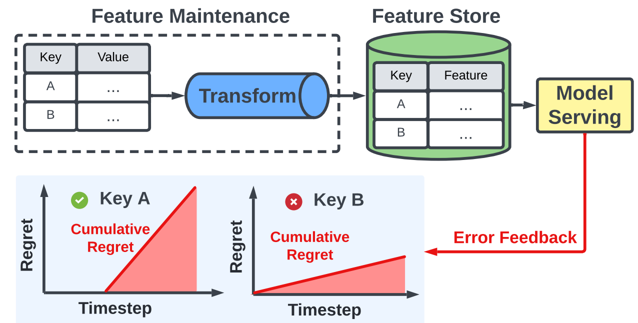

Figure 1: Feature stores serve materialized feature values to downstream models. RALF leverage downstream model feedback to prioritize expensive feature updates (§3.3.3).

Consider for example an online news recommendation service that predicts the probability that a specific user will click on specific article. Standard models for this task [31, 43] rely on sophisticated features (such as model based embedding) that summarize the user's click history, the text in the article, and the click histories of other users that have clicked on that article. These features are critical to making good predictions, but are expensive to compute and sensitive to the continuously changing news cycle.

Real-time model serving applications, such as online news recommendation services, require low latency predictions, and therefore rely heavily on pre-materialized *feature tables* stored and maintained by a *feature store* to hide the latency associated with deriving features. In order to provide low-latency access to important contexual information, *feature tables* At prediction time, the model serving system queries the pre-computed features from the feature store by specifying a *feature key* (e.g. a user ID), as shown in Fig. 1. However, because the features are often derived from data that is constantly changing (e.g., click streams and purchase history), the pre-materialized features also need to be continuously updated with the arrival of new data. Unfortunately, updating features with every data change can be wasteful and expensive for high-velocity data streams if the features are not read between updates or cannot be updated incrementally. Beyond computation cost, featurization via third party services also may impost hard rate limits on model-based embedding computations [6, 8].

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 17, No. 3 ISSN 2150-8097.
doi:10.14778/3632093.3632116

As a consequence, existing feature stores are faced with a choice between (1) greedily processing new updates as they arrive, and (2) allowing features to become arbitrarily stale. The former is often prohibitively resource intensive while the latter significantly degrades downstream model accuracy, as shown in Fig. 2. This trade-off is not unusual in this space: weakly consistent data stores are faced with similar issues. In general, relaxing consistency and allowing for stale data can break correctness in ways that are difficult to quantify [18, 46].

In the specific context of feature stores, however, "breaking correctness" has a measurable metric: *downstream model accuracy*. This is a clean metric that quantifies the prediction accuracy of a deployed model serving predictions. We can use downstream model accuracy as a guide for when and how to compute features and reframe the problem of building a resource-efficient feature store; rather than treating featurization as a task-agnostic data processing problem, we focus on maximizing downstream model accuracy.

We find that the appropriate feature maintenance policy for optimizing downstream accuracy can be key-dependent (within a single feature table) and vary across time. Keys that are rarely queried are unlikely to have significant impacts on overall downstream accuracy. Furthermore, even if keys are queried and updated at similar rates, the impact of staleness on accuracy varies dramatically by key. For example, some keys can be updated much less frequently than others without significantly impacting downstream accuracy, as show in Fig. 9. Prioritizing updates across keys can enable better resource efficiency in optimizing for downstream model accuracy.

In this paper, we introduce RALF (**r**eal-time, **a**ccuracy and **l**ineage-aware **f**eaturization) a feature store for real-time, high-density feature updates that explicitly leverages downstream feedback to reduce costs with minimal downstream accuracy degradation. We define a metric, *feature store regret*, to estimate accuracy degradation caused by featurization, and present feature update scheduling policies to minimize feature store regret.

Metrics. We argue the metric for evaluating a featurization pipeline should be based on downstream task performance. The ability to capture correctness numerically is a unique opportunity in striking the optimal balance between staleness, computation cost, and accuracy. Specifically, we define *feature store regret*, to measure the drift between the predictions made with optimal, high-cost features and predictions made with existing values in the feature store.

Propagating & Adapting to Feedback. RALF leverages knowledge of error feedback from downstream applications to estimate and minimize feature store regret in real-time. RALF achieves this by tracking the lineage between feature values and downstream predictions, and allowing downstream models to provide error feedback to RALF. This feedback allows RALF to prioritise recomputing the features that have the greatest impact on downstream accuracy.

To summarize, we make the following contributions:

(1) We formalize the feature maintenance problem and define a *feature store regret* metric to evaluate feature store state in terms of downstream accuracy.
(2) We introduce accuracy-aware feature maintenance policies to reduce the cost of maintaining features while also minimizing the feature store regret. We evaluate these policies with common feature store workloads, anomaly detection and recommendation.
(3) We implement a system, RALF, as real-time featurization pipeline that instantiates these policies. We evaluate RALF at scale with 257,077 keys for the anomaly detection workload to show up to 32.7% reduction in loss or 1.6× (i.e. 61%) compute reduction.

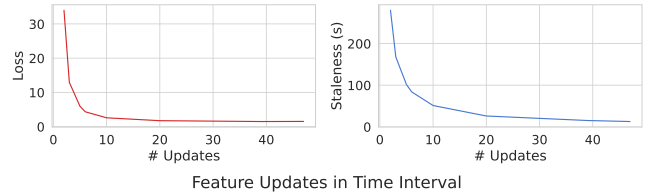

Figure 2: The prediction loss (measured by MASE) on the left is correlated with the feature staleness (time since last update), show on the right.

2 BACKGROUND

Most machine learning applications rely on *features* to summarize relevant aspects of the training data and provide the necessary context to make informed predictions. To illustrate, we return to our online news recommendation service example (§1).

In this setup, the model m must predict the probability $\hat{y}$ that user u will click on article a given a search query x. Most papers in the area will denote this seemingly simple prediction task as

$$\hat{y} = m(x, u, a). \tag{1}$$

After all, most papers in the machine learning and systems community are about how to design, train, and efficiently compute the model m. In this paper, we focus on how to compute the features, u and a, to optimize accuracy.

Hidden in this notation is the need to transform *historical data* associated with the user u and article a into their respective features, which can encode everything from the user's entire click history, to the contents of the article, and even the histories of other users that clicked on that article. As a consequence, a more accurate notation for this task would be:

$$\hat{y} = m\left(x, f_{\text{users}}\left(\mathcal{D}_u^t\right), f_{\text{articles}}\left(\mathcal{D}_a^t\right)\right), \tag{2}$$

where the functions f_{users} and f_{articles} are *featurization functions* and $\mathcal{D}_u^t$ and $\mathcal{D}_a^t$ are all the data up until the present (t) that is associated with the user and article. Each of the feature functions returns a vector that is combined with the query text x and processed by the model m. m makes a prediction, calculating the probability that user u will click the article a.

While the aforementioned example described only a couple of features, in practice, there may be dozens of features from different data sources computed for a single prediction. Automated feature generation tools make is easy to generate hundreds of unique features from data [3]. For notational simplicity, in this paper we will focus on a single featurization function f and key k:

$$\hat{y} = m\left(x, f\left(\mathcal{D}_k^t\right)\right). \tag{3}$$

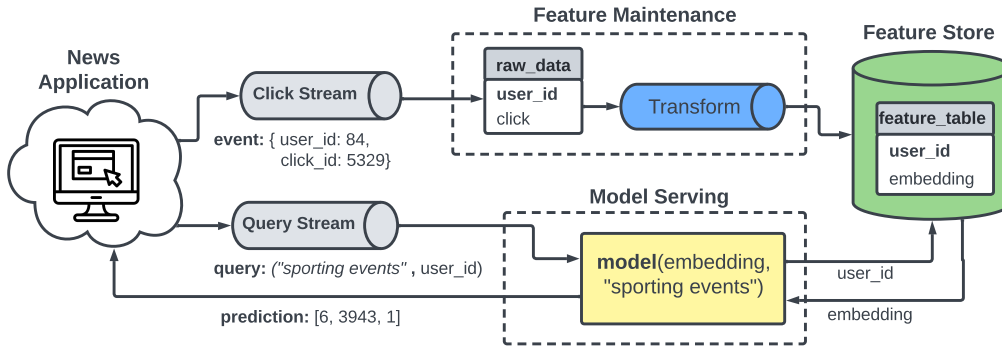

Figure 3: A ML serving pipeline with a feature store.

where x is the query and $\mathcal{D}_k^t$ is the historical data for key k.

Querying available historical data $\mathcal{D}_k^t$ and computing the featurization function f for each prediction request may be prohibitive in low-latency prediction serving settings where recommendations must be generated in real-time a users are scrolling through their news feed. Each query may need to access large amounts of historical data and run a computationally expensive feature function f. For example, many recent content recommendation models employ deep learning techniques to encode click streams and article contents and run online gradient descent [43].

Furthermore, many predictions may query the same keys, resulting in redundant computation. Often the same user features will be used to rank multiple articles and the same article will be ranked for many users. Executing the query on the entire history of users and articles for each new prediction is redundant, expensive, and infeasible for latency constrained settings.

To guarantee low-latency feature queries and avoid redundant computation, features are often *pre-computed* and stored in low-latency data store, referred to as *feature stores*.

2.1 Feature Stores

The *feature store* is a nascent class of systems which target the problem of storing and maintaining feature tables. We show an overview of a how feature stores, model serving, and applications interact in Fig. 3. There are several major open-source and commercial feature store systems [1, 2, 4, 9]. Feature stores can be used to fulfill a variety of requirements, such as enabling sharing of features across different multiple downstream applications, improving latency and cost by pre-computing features, and managing metadata about features (e.g. version control), which we discuss further in Section 7. In this paper, we focus specifically on maintaining feature table over streaming data updates in the context of online prediction serving.

As the underlying, raw data is updated over time, pre-computed features need to be *maintained* to prevent feature staleness, which may degrade prediction quality of dependent downstream models. For example, a feature encoding a user's interests in news topics can change rapidly with each new action by that users. If the feature is not updated over time, the stale encoding may degrade the quality of recommendations made by a model for that user.

Existing feature store typically rely on external data processing systems (e.g. Spark, Flink) to compute feature updates from new data. These systems then process new data in either a streaming or batch fashion to update current feature values.

2.2 Feature Maintenance

Maintaining features with new data can be computationally expensive, depending on the rate of new data arrival, the cost of the featurization function, and the required feature freshness. While some feature functions can be incrementally applied to new data, many require significant re-computation over a large window of historical data with the arrival of each new record. For example, an attention-based text document embedding model will need to re-compute the embedding of the entire document to reflect a single word change. Even when feature functions can be applied incrementally, running them in a streaming fashion on high velocity data streams can require expensive computational resources (e.g., GPUs) and be less efficient than large batch updates [20, 21].

Updating features with every data change can be expensive and unnecessary, depending on how quickly the true feature value is changing and how much impact staleness has on model predictions. In cases where models are robust to stale features, running a daily batch job to process new data is sufficient. In other cases where models are sensitive to feature staleness, features may need to be continuously updated with new data. For example, Splunk uses Flink for streaming maintenance of time-series features for real-time anomaly detection [41], and has developed application-specific solutions for maintaining fresh features for high cardinality data streams [41]. Feature values are typically *eventually consistent* with respect to the underlying raw data.

To provide a specific example, we implement a workload similar to Splunk's in Flink where we maintain a time-series decomposition for a set of cloud virtual machines, each streaming CPU utilization data. Updating a feature for a given virtual machine (i.e. the key) takes about 0.3 seconds, so a single Flink process can only update about 3-4 features per second. Existing systems do not natively have application awareness to prioritize updates, so will use a FIFO queue to process new data in incoming order. As a result, increasing the cardinality of the dataset eventually results in the

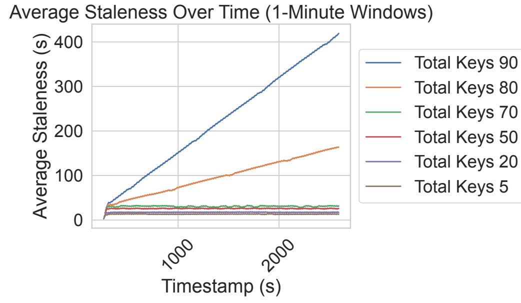

Figure 4: Average staleness in a 1-minute time window across all keys as a function as the cardinality.

per-key staleness linearly increasing with time as updates lag new data, as shown in Fig. 4. These increases in feature staleness are correlated to decreased prediction accuracy, as shown in Fig. 2.

In this paper, we show that scheduling feature updates according to each key's impact on downstream accuracy allows us to preserve overall accuracy at lower cost. Feature stores typically lack awareness of downstream query patterns and performance of the predictions made using queried features. As a result, systems for maintaining features treat all data updates and keys symmetrically and fail to leverage important information about which updates are critical and which keys are likely to be accessed in the downstream prediction workload.

2.3 A Feature Store Reference Model

For simplicity, we first describe the standard formulation of a feature store. In Section 7, we discuss the full variety of feature stores presently being used, and how our work applies.

We assume that raw historical data is loaded into a data warehouse, capturing the basic entities (users, movies) and actions (users seeing ads, users viewing movies, etc) we use in prediction. We then consider a derived *feature table* that memoizes *featurization functions* over that data. This table can also be stored in the data warehouse, or it can be maintained in an external cache database like Redis or Memcached; our design does not depend on that decision. A SQL query that populates the feature table exhaustively would have a template that looks like this:

```
SELECT key, uda(data)
FROM historical_data
GROUP BY key
```

where uda is a user-defined aggregate function. If the feature store is kept in the warehouse, feature tables can be viewed as traditional materialized views. Materialized views, however, are typically kept consistent with underlying data, and must be recomputed on every new update. Systems that support incremental view maintenance incur similar costs when the supplied feature function cannot be recomputed incrementally. In contrast, RALF focuses on carefully choosing when and what keys to recompute to minimize resource costs while preserving accuracy:

```
SELECT key, uda(data)
FROM historical_data
WHERE key IN <PolicyQuery>
GROUP BY key
```

The fundamental policy decision addressed in this paper is: *given the above query can only be run on a small subset of all possible keys at a time , which keys do we select to ensure maximum downstream prediction accuracy.* We focus on making scheduling decisions across keys (rather than between updates pertaining to a single key), as large key cardinality is a common attribute in feature store applications. We use SQL here to illustrate our ideas, but of course this logic could be implemented in a number of scalable data-centric APIs, including Spark, Flink, and so on.

As we discuss in Section 7, there are many options for materializing and storing features. Our simple model here is designed to be sufficient to illustrate the key policy issues at hand; further architectural complexity is discussed in Section 7.

3 EFFICIENT FEATURE MAINTENANCE

In this section, we formalize the feature maintenance problem addressed in this paper, that is, selecting the keys for §2.3. In a resource-constrained setting, only a subset of features can be updated at any given time, resulting in feature staleness which may degrade prediction accuracy. The focus of this paper is precisely to optimize this issue: deciding which keys to update in response to new data, with the objective of maximizing downstream prediction accuracy. As previously highlighted, the core enabling factor is the differentiated impact that feature staleness has on overall accuracy: stale features may lead to low query errors, while some features may simply rarely be queried at all.

3.1 Feature Approximation

Featurization cost can be reduced by computing features using approximated featurization (e.g. sampling) or using stale features, which is the focus of this paper. Reducing the frequency of updating feature values by tolerating staleness is a simple way to reduce featurization cost, as the same update function can be used on the same data: the only parameter to change is when the update is triggered. For example, multiple edits to a document can be batched together so the document only needs to be re-embedded once, or a function over a window of data can be run less frequently to reduce computational cost.

For feature derived from data $\mathcal{D}^t$, we denote the true feature values at time t as $v_k^t = f\left(\mathcal{D}_k^t\right)$, and the stale feature values as

$$\tilde{v}_k^t = f\left(\mathcal{D}_k^{t-\delta_{k,t}}\right). \tag{4}$$

where $\delta_{k,t}$ is the staleness of the current feature value. Delaying update processing, and thereby increasing the staleness, reduces how often f needs to be run on new data. However, reducing the frequency of re-computation results in features that are more stale, as entries in the feature table are more likely to be missing the most recent updates.

3.1.1 *Evaluating Approximation Quality.*

Standard ways to evaluate the quality of approximation is to evaluate the staleness of the queried data, or the differences in the approximated and unapproximated value. However in the context of feature stores, these metrics do not necessarily correlate to prediction quality. Feature

staleness or large divergence in feature values is not problematic if the prediction quality is not impacted. Similarly, slight changes in the feature values can dramatically change predictions. For example, neural networks can be very sensitive to small perturbations in input, and it is difficult to model how differences in feature values will correlate to differences in predictions, especially when the input values to the model are unknown.

However, directly using downstream accuracy as a metric for feature quality is problematic, as prediction quality depends on *both* the features and the model. A model may perform poorly for an out-of-distribution user regardless of feature approximation quality. In order to disentangle model performance from feature quality, we propose *feature store regret* in the next section.

3.2 Feature Store Regret

We propose a feature store metric, *feature store regret*, to evaluate feature quality. The feature store regret is the difference in predictions made by the optimal feature values v_k^t and approximated features $\tilde{v}_k^t$.

$$\mathcal{R}(t) = \mathcal{L}(m|\tilde{v}^t) - \mathcal{L}(m|v^t) \tag{5}$$

where $\mathcal{L}(m|\tilde{v}^t)$ and $\mathcal{L}(m|v^t)$ are the *total loss* of predictions made with the approximated and unapproximated feature values, respectively. For simplicity, we assume $\mathcal{L}(m|\tilde{v}^t) \geq \mathcal{L}(m|v^t)$. We can write the total loss in terms of the sequence of prediction requests with data $\{x_i\}$ at time t which correspond to predictions $\hat{y}_i(v_t)$ and true values y_i:

$$\mathcal{L}(m|v^t) = \sum_i \ell\left(\hat{y}_i(v_t), y_i\right) \tag{6}$$

where ℓ is the loss function used to evaluate the model.

3.3 Scheduling with Error Feedback: Regret-Proportional Scheduling

We propose an online scheduling policy in cases where we can observe regret online, which we refer to as *Regret-Proportional* update scheduling. In many model serving applications, the true prediction label can eventually be observed. For example, a recommendation model can serve recommendations to a user and eventually observe which recommendations the user did or did not click through. Similarly, a time series feature can be evaluated against future points observed for the time-series. The observations of the true label can be used to compute model prediction error, which can be used to provide feedback on feature quality. While prediction error cannot always be computed online, we constrain the problem to this setting to consider how error feedback can be used to make better scheduling decisions.

We formalize the online scheduling problem for feature stores in terms of minimizing feature store regret under resource cost constraints. At a high level, our proposed policy is to prioritize keys with the highest cumulative regret. This allows us to prioritize updating keys where feature staleness has the highest impact on the overall loss rather than keys where the prediction loss is primarily a result of model error. We describe how we estimate regret with error feedback in §3.3.3.

3.3.1 Formulation. We consider a feature table with keys $k \in K$ each mapping to values $\tilde{v}_k^t$. At time t, the scheduler can update a subset of keys $U_t \subseteq K$. For each $k \in U_t$, we recompute the feature value on all data up to the current timestamp, while other feature values remain the same. We can denote the approximate feature values at time t with Eq. (4) where the staleness $\delta_{k,t} = 0$ if the key k is updated at time t, and otherwise $\delta_{k,t} = 1 + \delta_{k,t-1}$.

Given a constraint C on the number of keys which can be updated at each timestep t, our goal is to select updates U such that the staleness matrix δ minimizes the cumulative regret over time:

$$\arg\min_{\delta} \sum_t \mathcal{R}(t) \tag{7}$$

$$|U_t| \leq C, \forall t \tag{8}$$

3.3.2 Error Feedback. We assume that we can observe the per-key loss. Say that for the sequence of queries $\{x_{kt}\}$, we eventually recieve error feedback $E_t = \{e_k\}$ denoting the prediction error of $m(x_{kt}, \tilde{v}_k^t)$. For simplicity, we assume that the error is received before we need to make scheduling decisions for the next timestep. We can estimate the per-key loss at each timestep as the sum $\mathcal{L}(m|\tilde{v}_t^k) \approx \sum_{e_k \in E_t} e_k$.

3.3.3 Scheduling Policy. We propose an online algorithm which selects keys to update based off the cumulative regret observed since the last update:

$$\arg\max_k \sum_{s=0}^{\delta_{t,k}} \mathcal{R}_k(t-s) \tag{9}$$

To estimate $R(t)$, we also need an estimate of the loss with the ideal features $\mathcal{L}(m|v_t^k)$. We assume that the *expectation* of error over queries is temporally stable with respect to staleness for each key. Thus we can calculate the average error immediately after the feature was updated at time $t_u = t - \delta_{t,k}$ and multiply with the number of error observations at time t to estimate $\mathcal{L}(m|v_k^t)$ and subtract this from each error value observed at timestamp t before taking the sum of all errors observed at t. We can thus write out the estimated regret at t as:

$$\mathcal{R}_k(t) \approx \sum_{e_k \in E_t} e_k - \sum_{e_k \in E_{t_u}} \frac{|E_t| \cdot e_k}{|E_{t_u}|} \tag{10}$$

Intuitively, we can think of this as computing how much additional error per query there is in E_t (the current timestep error) as compared to E_{t_u} (the post-update timestep error). Expanding out Eq. (9) and denoting the last update time as $t_u = t - \delta_{t,k}$, we select the key to update as:

$$\arg\max_k \sum_{s=0}^{\delta_{t,k}} \sum_{e_k \in E_{t-s}} \left(e_k - \sum_{e_k \in E_{t_u}} \frac{e_k}{|E_{t_u}|} \right) \tag{11}$$

We can prevent starvation by upper bounding the regret $R_k(t) < \mathcal{R}_{max}$, and assuming $R_k(t) > \epsilon$ for some $\epsilon > 0$. We find in practice, since the errors in E_{t_u} are relatively small, we can remove the second summation term and simply sum e_k to estimate regret.

3.3.4 Default Regret. One potential issue with relying on cumulative regret for key prioritization is that a key may become arbitrarily stale if the key is never queried. Stale keys can be prioritized more by setting a higher minimum regret value $R_k(t) > \epsilon$, so that keys will incur regret over time.

Listing 1: Defining a maintained feature table of user embeddings with RALF.

```
# Source table
source = ralf.tables.kafka_source(topic="user_data")

# Queryable feature table
embedding = source
    .map(UserEmbeddingModel, model_file="model.pt")
    .as_queryable("user_features")
    .set_replicas(4)
    .set_default_error(0.01)
```

Listing 2: Example of a downstream application serving predictions using queried feature values and posting error feedback once the result is observed.

```
class CartAbandonmentModel:
    client = ralf.client(table="user_features")
    cache = {}

    # serve prediction requests
    def predict(user_id, cart_id):
        feature, fid = client.get(user_id)
        cache[cart_id] = {
          "pred": model.predict(feature, cart_id),
          "feature_id": fid,
          "feature_key": user_id
        }
        return cache[cart_id]

    # post feedback when label is received
    def on_label(cart_id, checkout: bool):
        error = MSE(cache[cart_id]["pred"], checkout)
        client.feedback(
          key=cache[cart_id]["feature_key"],
          feature_id=cache[cart_id]["feature_id"],
          error=error
        )
```

4 SYSTEM DESIGN AND ARCHITECTURE

In this section, we describe RALF, which orchestrates updates to feature tables with adaptation to feedback. Downstream clients query the feature tables through the RALF client so that RALF can track query access patterns and also post feedback to RALF once prediction labels are observed.

4.1 RALF Server

RALF orchestrates data updates to maintain feature values. We show an example of defining a maintained feature table with RALF in Listing 1. RALF schedules and processes data updates to compute new values for the feature table using the specified feature transformation. In addition, RALF receives queries and error feedback from the client in order to track feature access patterns and quality. RALF requires a feedback loop: a downstream model that queries feature values must post the observed error for the corresponding

---

Algorithm 1: Choosing next key to update

Data: List of feedback $F[k]$ for key k, $pendingKeys$, $processingKeys$
Result: Selected key k to process updates for next.
$chosenKey \leftarrow -1$;
$maxRegret \leftarrow -1$;
for $k \in pendingKeys$ **do**
  $regret = F[k].sum()$; `/* Calculate regret */`
  if $regret \geq maxRegret$ **then**
    $maxRegret \leftarrow regret$;
    $chosenKey \leftarrow k$; `/* Update chosen key */`
  end
end
$F[chosenKey] = []$; `/* Clear key feedback */`
$pendingKeys.remove(key)$;
$processingKeys.append(key)$; `/* Key is processing */`
return chosenKey

---

key back to the server. This data is used by the scheduler to help decide which key to update next.

4.1.1 *Transformation.*

Feature transformations are defined by user definted functions (UDFs) which can maintain state and define an *on_event* function, which define how to transform a data update from the raw data table to a data update to the feature table. We show an example transformation in Listing 1. These transformations are implemented as Ray actors, so RALF relies on Ray for concurrency and fault tolerance.

4.1.2 *Scheduling.*

Pending updates are scheduled by RALF with the scheduler, which chooses the next key to update. The scheduler receives error feedback from downstream models, and uses this to update a table tracking estimated cumulative regret per key. This table is used to select the key with the highest estimated regret. The chosen key and corresponding data passed to the transformation.

4.1.3 *Scaling.*

RALF scales to large cardinality datasets by sharding keys across multiple replicas, which each replica can run on separate processing across a single or multiple machines. Each replica has a separate scheduler and error table to avoid coordination.

4.2 RALF Client

The RALF client is used by downstream applications to query RALF for features and to post feedback. We show an example of a downstream application in Listing 2, which queries the client for feature values to predict the likelihood of cart abandonment. For applications where true labels are later provided, the application can also post feedback to the client to inform future scheduling the decisions. The feedback takes in the key of queries feature, the queried feature version, and the error of the resulting prediction. Feedback is posted to RALF, which tracks error feedback for current feature versions on the feature view.

4.3 Scheduling Policy

RALF schedules feature updates with Regret-Proportional scheduling, that is, prioritizing updates to keys with the largest *cumulative*

regret. The cumulative regret is calculated by tracking the reported error for predictions made using the current feature version stored in the table, and then selecting the key with the largest cumulative regret (as shown in Algorithm 1). Once a key is chosen, the prior feedback and queue for the key are both cleared, and the key is marked as being processed and locked until the new feature value is computed. The scheduler tracks a list of *pendingKeys*, the list of keys with new data updates, and *processingKeys*, the list of keys where new features are currently being computed. Keys are selected from *pendingKeys*, and once selected, are removed from *pendingKeys* and added to *processingKeys*. Keys in *processingKeys* cannot be chosen again by the scheduler until they are removed once the featurization update is complete - this is to prevent duplicate updates to keys while they are still processing.

4.4 Implementation

We construct a full prototype of RALF in about 1,500 lines of Python code. Our prototype is built atop Ray [42], because many popular featurization and machine learning libraries (e.g., Tensorflow [11]) use Python, and Ray is designed to support machine learning workloads. We emphasize that RALF is a set of ideas for accuracy-aware featurization, and can be implemented on several systems.

5 EVALUATION

In this section, we address two primary questions: (1) how does Regret-Proportional scheduling impact downstream prediction accuracy (2) how does RALF with Regret-Proportional scheduling scale to processing high-cardinality, high-rate data streams? To answer these questions, we structure our evaluation as following:

(1) We construct representative workloads for two common feature store use-cases, recommendation and anomaly detection, using real-world datasets. For both workloads, we evaluate feature quality by evaluating model predictions that rely on feature which are updated over time.
(2) We run an end-to-end evaluation with RALF on a large-scale anomaly detection workload to evaluate prediction accuracy improvements, system overhead, and scaleability.
(3) We run ablations comparing Regret-Proportional scheduling with both baseline and application-specific policies.

5.1 Workloads

To evaluate feature maintenance policies, we construct workloads using real-world data where model predictions rely on pre-computed features that need to be updated as new events are streamed in. For each workload, we use real-world data to generate an *update stream* (incoming raw data), *query stream* (queries from downstream models), and *feedback stream* (error feedback).

For each workload, we setup to following components to mimic realistic prediction serving applications: A **feature function** (the operator that transforms data streams into features cached in the feature table), **feature table** (the key/value store contained feature keys and values), and **downstream model** (the downstream prediction serving application which queries feature table values that are used to make predictions).

We describe the dataset, featurization, and downstream models for recommendation and anomaly detection workloads. A summary of workload attributes is show in §5, which also shows the best and worst-case prediction loss depending on feature quality.

5.1.1 *Anomaly Detection.*

Time series decomposition is a common pre-processing step to many downstream tasks, such as anomaly detection or forecasting. We construct an time-series anomaly detection workload based off a real-world application at Splunk [41, 48]. The anomaly detection task compares predicted points from time-series features, calculated from windows of past data, with the observed points to detect anomalies. Accurate anomaly detection depends on estimating the residual of the point accurately, which relies on the accuracy of the cached time-series features. The *query stream* periodically queries all keys to detect anomalies in regular time-intervals, so the distribution of queries over keys is uniform. Features are maintained over an *update stream* of new time-series points. Each new time-series point is compared to previously predicted points to provide a *feedback stream*.
Dataset. We use both the Yahoo Webscope S5 Dataset's A1 class [34] and Azure VM dataset [19].We use the Python statsmodels library [45] to compute features from windows of data for each time-series. For the Yahoo dataset, the rate of updates and start time for each time-series is uniform across keys, so the distribution of queries and feedback across keys is also uniform. However, the variation in the time-series can vary dramatically across keys, opening opportunity for optimizing resource allocation across keys. For example, some time-series vary little over time, while others change rapidly and have complex and variable seasonality components.

5.1.2 *Recommendation.*

Recommendation is another applications where machine learning models are used to make low-latency recommendations to users, often using user features derived from historical data to personalize predictions. We construct a recommendation workloads where a downstream models predicts what a user's rating for a movie will be user and movie features computed from past rating data, where user features are updated online. Given a stream of user ratings for movies, we simulate a *query stream* over the users to predict what the rating should be. We return the prediction error of the rating as the *feedback stream*, and treat the rating itself as data update from the *event stream*. The incoming event stream of ratings is used to update user embeddings over time using partial ALS to update the corresponding feature vector.
Dataset. We use the MovieLens 1M [5], which has timestamped ratings from roughly a million user/movie pairs. We use the first half of the data to train a model using Alternating Least Squares. We treat the resulting movie embeddings as the static *model* and the user-ratings as *features* which are updated over time. We use the second half of the data as query, event, and feedback streams.

5.2 End-to-End Evaluation

We evaluate RALF on 800 cores for end-to-end with the Anomaly Detection workload using the Azure VM dataset [19]. We run RALF with both our Regret-Proportional policy and baseline policy of Round-Robin scheduling to evaluate prediction accuracy, scheduling overhead, and scaleability.

5.2.1 *Experimental Setup.*

The Azure VM dataset includes of the CPU readings taken every 5 minutes on a pool of 2 million VMs over the span of one month. We send a subsample of 275,077 time-series

Table 1: Workload attributes. The *Runtime* column refers to the featurization update runtime for a single key. The *Min Loss* and *Max Loss* columns show the overall loss given infinite budget and zero budget for featurization, respectively. The minimum loss for the Azure dataset is shown in Fig. 5.

Workload	Dataset	Keys	Runtime	Edits	Min Loss	Max Loss
Recommendation	MovieLens 1M	6041	0.9s	85,297	1.12	6.29
Time-Series	Yahoo Anomaly A1	68	0.25s	43,684	90.79	880.3
Decomposition	Azure VM Dataset	275,077	0.4	5,683,390	-	-

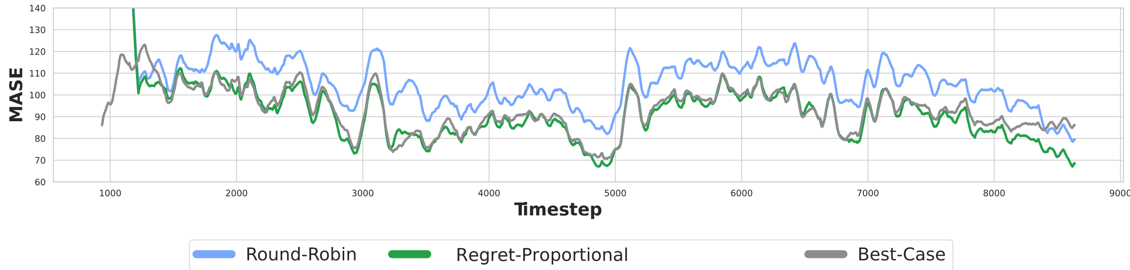

Figure 5: Smoothed Average MASE per Timestep over 275,077 keys.

from Azure Dataset on a cluster of 11 m5d.24xlarge machines (800 cores) on AWS. We simulate higher data send rates by sending at 1000x speed (i.e. ingesting data once every 0.3 seconds, rather than every 5 minutes as specific in the dataset). We use RALF to compute an STL decomposition of the time series for each key, using a recent observation window. We set the STL decomposition seasonality to be 24 hours, and set the observation window size of data to be 3X the seasonality length (so 72 hours of recent data points) to have a sufficiently large window to compute the decomposition. We store the resulting STL decomposition as a feature in the feature store for each key (i.e. a time-series ID), which is updated over time by RALF as new data arrives. Because of the high data rate, some features will be out of date with the current observation window. RALF uses either the Regret-proportional or Round-Robin scheduling policy to choose which features to prioritize updating.

5.2.2 Policy Error. To evaluate feature quality, we compare the MASE (Mean Absolute Squared Error) of time-series predictions using the STL decomposition features using the Regret-Proportional and Round-Robin scheduling policies in Fig. 5. We can calculate the MASE by comparing the predicted points with the actual points observed. We show a plot of average MASE across keys over time for features computed with the Regret-Proportional and Round-Robin scheduling policies in Fig. 5. Although overall MASE varies over time, the Regret-Proportional policy consistently produces lower MASE than the Round-Round policy features, with error improvement ranging from 2-32.7% and averaging to 13%.

We additionally calculate the *optimal* version of the features (described in §3.2) for each query by calculating what the feature value would be with all data up to exactly the query time. The optimal features correspond to the best case MASE (shown in grey in Fig. 5) enabled by unlimited compute resources (i.e. processing every possible update). We see that the MASE for optimal features

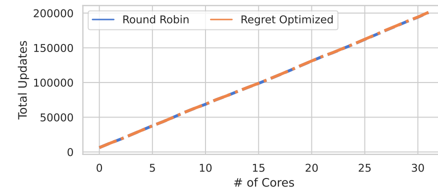

Figure 6: System throughput versus number of cores.

and the Regret-Proportional policies are similar in Fig. 5. The Regret-Proportional policy over the course of the experiment runs 61% fewer updates (i.e. 1.6× less) than would be needed to achieve the optimal features, however averages only 1% additional error as compared to optimal features.

5.2.3 Scaling Evaluation. We evaluate how RALF's throughput scales in Fig. 6 by measuring the throughput per number of cores for Round-Robin versus Regret-Proportional scheduling. For both the Round-Robin and Regret-Proportional policies, the throughput scales linearly with the number of cores. Because the workload is embarrassingly parallel, we can shard keys across replicas, where each replica corresponds to one core and has its own scheduling and transformation operator. As a result, the number of updates scales linearly with with the number of cores. We use randomized hashing to place keys on replicas and utilize 800 cores of workers.

5.2.4 Scheduling Overhead. We evaluate the scheduling overhead of Regret-Proportional versus standard Round-Robin scheduling in terms of both compute and memory. The Regret-Proportional policy requires a constant CPU cost of 300 μs per arrived window queued for update in order to evaluate the regret score. Furthermore, maintaining a sorted queue (ordered by per-key regret) costs

50 μs per addition/removal. Additionally, because the regret calculation requires previous feature to be cached in memory, the Regret-Proportional policy also costs about 32 KBs per key, resulting in about 11MB of memory overhead per core. We note that the per-core compute and memory overhead is constant regardless of the number of cores used, due to scheduling occurring per-replica rather than globally. This allows us to mitigate coordination overhead and is sufficient for making scheduling decisions that load balance across threads and optimize feature quality.

We plot the total throughput as a function of total cores in Fig. 6. The Regret-Proportional policy performed 0.6% less updates as compared to Round-Robin policy. However, despite fewer number of updates performed, the cached features from Regret-Proportional scheduling results in significantly better model performance. This is because the Regret-Proportional policy can achieve similar feature quality with dramatically fewer updates, as shown by achieving near-optimal feature quality with 61% fewer updates.

5.3 Policy Ablations

We compare the Regret-Proportional scheduling policy to other baseline and application-specific policies that do not consider downstream prediction accuracy. We run simulated experiments with both the Recommendation workload and the Anomaly Detection workload (using a smaller time-series dataset, the Yahoo A1 dataset) to show the generality of our policy improvements.

5.3.1 Policies. We implement the **Regret-Proportional** policy described in §3.3. Similarly, we implement a **Query-Proportional** policy which updates features proportionally to the rate they are queried (i.e. the number of times the feature has been queried since last updated), to understand the impact of regret versus query awareness. We evaluate these policies along with baseline query-oblivious policies commonly found in stream processing systems.

We implement baseline query-oblivious policies for choosing which keys to update:

- **Round-Robin**: Iterate over each key and skip keys with no pending updates (equivalent to updating the most stale and least-recently-updated key).
- **Random**: Randomly select a key with pending updates.

We additionally implement two more sophisticated query-oblivious policies designed to improve accuracy in the Recommendation workload:

- **Minimum-Past**: Update keys that have the least data incorporated into the feature (i.e. the number of ratings seen for the user).
- **Max-Pending**: Update keys with the most pending new data (i.e. the user with the most new ratings).

5.3.2 Prediction Error . To evaluate the quality of features derived with different policies under different cost constraints, we simulate the policies for each workload. At each timestep in the simulation, there is a set of feature update events and feature queries for a set of prediction at that timestep. We set an update budget, which limits the number features we can update per timestep. The subset of features to update is chosen by the scheduling policy. At each timestep, the simulator processes some subset of feature updates chosen by the scheduler and generates predictions using the current set of features, which we use to evaluate error in Fig. 7.

5.3.3 Regret-Proportional Policy. The Regret-Proportional policy is able to achieve better error across different workloads and numbers of udpates, as shown in Fig. 7. Query-Proportional updates improves error over baseline policies for the Anomaly Detection workload, as shown in Figure Fig. 7. However for the Recommendation workload, where it is crucial to update features with little prior data (e.g. new users), the updating proportionally for the queries fails to account for the non-uniform benefit of updates across features. As a result, the Minimum-Past policy, which updates the feature with the fewest prior updates, significantly outperforms the Query-Proportional policy for Recommendation. Weighing the queries by the regret they incur (as in the Regret-Proportional policy) improves the results beyond Query-Proportional updates alone by accounting for *both* the query pattern and the significance of updates.

New users who have no associated ratings (and hence very poor quality default features) are prioritized by Minimum-Past and Regret-Proportional policies, which significantly improves performance over other policies. However, Minimum-Past cannot distinguish the important of updates between users with similar prior update histories, resulting in worse performance than Regret-Proportional overall. We measure the MSE improvement from the Regret-Proportional policy over Minimum-Past for users with past ratings (Trained) versus new users (Untrained) in Fig. 8. Although both policies are similar for new users, the Regret-Proportional policy has substantial improvements over Minimum-Past for existing users keys. The Regret-Proportional policy is able to account for the importance of prioritizing updating new users' features, while also intelligently prioritizing updates across users for which features have already been computed.

5.3.4 Distribution of Updates. Different policies allocate update budgets in different ways across keys. The variation is most clearly observed for the Anomaly Detection workload, where keys have raw data updates and queries arriving at uniform rates, but are updated with very different distributions depending on the policy, as show in Fig. 10. The Regret-Proportional policy is able allocate more updates to features incurring regret the most rapidly, resulting in large update variations.

5.3.5 Optimizing Feature Staleness versus Feature Quality. Although the staleness of the features is correlated to the prediction accuracy as shown in Fig. 2, we find that the best performing policies in terms of prediction error are not the best performing in terms of staleness data. As shown in Fig. 10, the Regret-Proportional has higher average staleness than other policies, including Round-Robin. This is because other policies such as Round-Robin will always prioritize updating the most stale feature, rather than the most important feature to update to optimize downstream prediction error. As a result, the Regret-Proportional policy results in better prediction error despite increased staleness, as shown in Fig. 7. Although staleness is strongly correlated to feature quality, optimizing for staleness does not always have the same results as directly optimizing for feature quality.

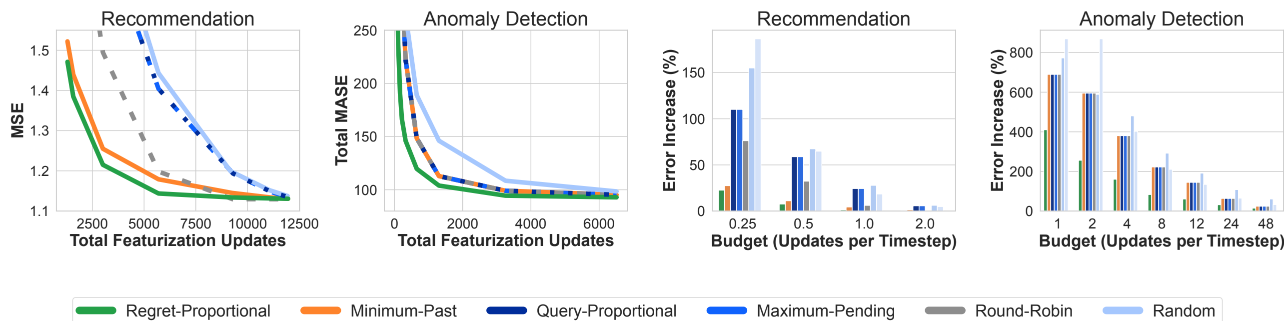

Figure 7: Left: Prediction error versus total featurization updates (over the entire experiment). Right: Error increase (compared to optimal features with unlimited budget) for varied update budgets.

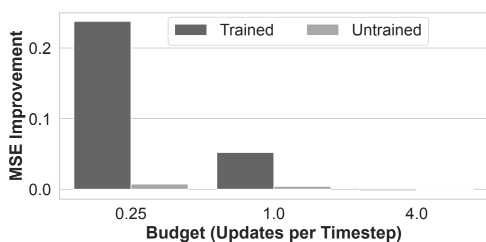

Figure 8: Regret-Proportional Improvement over Minimum-Past for users in the training set (Trained) versus new users (Untrained) in the Recommendation workload.

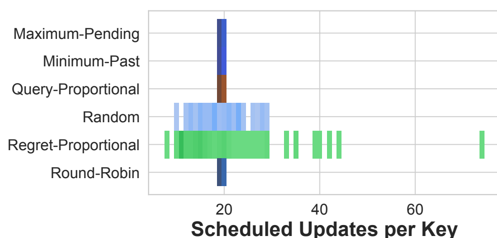

Figure 9: Distribution of featurization updates (Anomaly Detection): The Regret-Proportional policy has the most variability in number of updates per key.

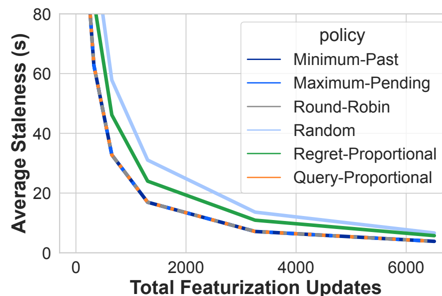

Figure 10: Queried feature staleness (Anomaly Detection): Average staleness at query time across key, measured by the number of timesteps since the last update.

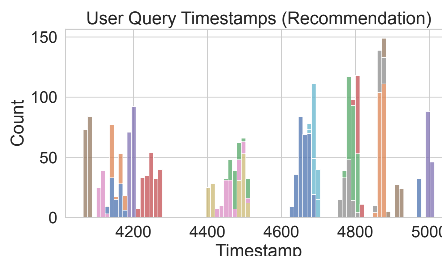

Figure 11: Frequency of user queries over timestamps.

5.3.6 Query Distributions. The Anomaly Detection workload has a uniform query distribution over keys, while for the Recommendation workload, queries for a given user typically come in bursts after long periods of inactivity. We additionally test the effect of different query distributions by re-assinging the inter-arrival times for the Recommendation workloads. We re-assign the inter-arrival times between events to follow an Exponential distribution (equivalent to a poisson process) and a Gaussian distribution, where the mean inter-arrival time is the same as the original distribution. We show in Fig. 12 that this leads to similar results as the original distribution of data, showing that Regret-Proportional scheduling is robust to different query distributions.

5.4 How well can future error be predicted?

We evaluate how well error from past queries can predict errors in future queries as a function of the window size of the past queries considered and the lag between the error data and timestamp which we are trying to predict error for (which we refer to as the offset). We train a linear regression model on both the Recommendation and Anomaly Detection workloads to predict error for a future timestep (with some offset) given a window of previous errors for a given key. We show results in Fig. 13, where we plot the MSE of the predicted error. Both workloads benefit from larger windows, but is especially important for the Anomaly Detection workload. Varying the offset hurts the accuracy of the model in Recommendation, suggesting that the freshness of the feedback is

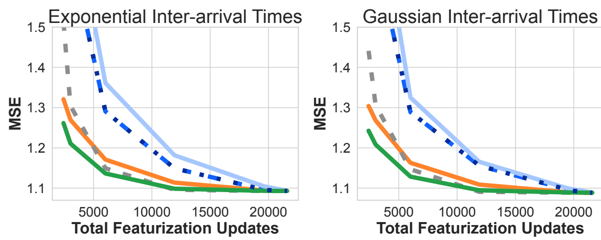

Figure 12: Recommendation workload with modified query inter-arrival time shows similar results.

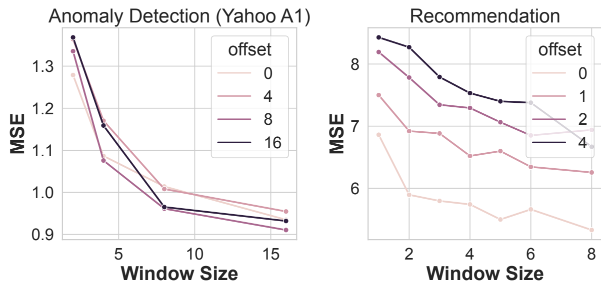

Figure 13: Predicting next timestep error for different window sizes (number of past errors) and offset (timestep delay of the window).

critical, while Anomaly Detection relies on just having a sufficient window size (since the per-key error is much more stable over time).

5.5 Regret-Proportional Scheduling Limitations

In our workloads, we assume that that the prediction error can be observed and fed back to the scheduler; this allows us to make decisions that will minimize future prediction error by selectively updating certain features. Our purpose in this evaluation is to demonstrate that such feedback from downstream applications—providing recent prediction errors and query patterns— can be leveraged to make better scheduling decisions for feature maintenance. We believe that future work will be able to make progress is learning to effectively estimate regret from offline data for certain workloads.

There is additionally a concern here with coverage. If we only update features that have incurred past regret, we will fail to update features that have not been queried in the past (e.g. a user who has not logged in in a long time suddenly begins a new session). Such keys form the long tail of the query distribution. To handle this concern, RALF can be used with a higher default regret value (described in §3.3.4, which will ensure that sufficiently stale keys will eventually be prioritized. However, even without this, since RALF's policy is online, so can react quickly to prioritize features that suddenly start to get quried, as shown in our results from the Recommendation workload.

6 RELATED WORK

Feature Stores. While industry has heavily adopted the use of feature stores [4, 9], academic research on these systems is limited, and remains focused on metadata and lineage management [28]. We discuss feature stores in depth in §7.

Approximate Query Processing. Approximate query processing reduces the cost and latency of queries by returning approximate results [12, 16]. In the machine learning context, recent work investigates how cheaper and more expensive models can be combined to respond to queries [29] while providing formal bounds on approximation [30]. RALF focuses on minimizing how frequently feature computation takes place, not on minimizing computation costs. Approximate query processing could be used in conjunction with RALF to target the latter. Investigating how these two approaches interplay is a promising avenue for future work.

Materialized View Maintenance. Feature tables can be thought of as a materialized view [17] over raw data sources. Prior work in incremental view maintenance and partial view maintenance [52] have examined how to efficiently maintain views over changing data. Noria [24] uses partial state and eventual consistency to efficiently materialize tables both on events and on queries. Timely Dataflow [10] leverages shared arrangements [40] to facilitate incremental recomputation of a view. No existing work focuses on *when* to recompute a given view and how to prioritize across view to optimize application correctness.

Prediction Serving. Most prior work in prediction serving [20, 21] focuses on optimizing model serving resource efficiency but does not consider the feature stores in such pipelines; these systems exclusively target improving model inference and fail to consider data preprocessing and featurization. Systems that do consider featurization either focus on making use of cheaper featurization functions which can be approximated without affecting prediction [32], or target specific application use cases such as video analytics [15, 27, 38].

Staleness and Consistency. Trading-off consistency for performance is a well-known strategy in modern large-scale distributed systems. These key-value stores or databases relax constraints on when and how operations must take effect, reducing the cost of synchronization [14, 18, 23, 39, 39, 46, 47, 51]. The flip-side is the increased programmer burden in defending against the potential unexpected application behaviours that arise from these relaxed guarantees [14, 22]. To minimize this issue, prior work either 1) distinguishes between operations whose ordering can be safely relaxed [13, 33, 37] 2) bounds divergence from the true value when possible [23, 26, 49–51]. The former is often restrictive, while the latter does not discuss the application-level consequences of diverging from the true value. These limitations have led to skepticism as to whether weak consistency is valuable option for developers. Feature store systems, in contrast, have *explicit* metrics and mechanisms to understand loss of correctness; they are thus uniquely positioned to leverage weak consistency and the staless/consistency tradeoff that it enables.

7 DISCUSSION

In this paper we focused on feature stores in the context of online serving and real-time maintenance for staleness-sensitive features. However, real-world deployments of feature stores have diverse requirements, design choices, and applications in ML pipelines.

7.1 Feature Materialization

Most feature stores do not support feature materialization, and instead support ingestion of pre-computed features through streaming and batch ingest pipelines. Other feature stores (e.g. Tecton) offer built-in transformation tools. Existing feature transformation systems are usually built on top of multiple existing computational engines (e.g. Flink, Spark, AWS lambda) to support different ways of materializing features, making it difficult to apply general optimization techniques across them.

For feature stores that support materialization (e.g. Tecton), there are typically three types of feature materialization:

(1) **Batch**: Features are periodically refreshed (e.g. daily, hourly) with a batch processing system (e.g. Spark, Airflow).
(2) **Streaming**: Features are continually re-computed with new data arriving in a streaming fashion with a streaming system (e.g. Flink, Spark Streaming).
(3) **On-Demand (i.e. Lazy)**: The feature is materialized at query time (e.g. with a lambda function).

Whether feature should be pre-materialized or materialized in real-time depends on the cost of featurization, the query latency requirements, and the rate of incoming new data and queries. Batch feature updates can be more cost effective for features which are not staleness sensitive. On-demand feature updates are cost effective when the latency of the feature computation is very low or there is not a requirement for low-latency queries.

Although we primarily focused on the case of steaming materialization of expensive features, the policies in RALF can be applied to any case where only a subset of keys can be processed. This applies to both streaming materialization and batch materialization where the throughput may be limited.

Prior work in approximate query processing and approximate featurization [32] can also reduce the cost of feature materialization and be used in conjunction with batch, streaming, or on-demand materialization. We consider this line of work orthogonal, as both key-prioritization and feature approximation can be combined to reduce cost.

7.2 Feature Storage

Feature stores are typically responsible for serving features to online model serving pipelines with low latency, as well as storing large amounts of historical data and features for model training pipelines. As a result, feature stores typically contain two separate datastores: 1. an *offline store* for offline training, and 2. a *online store* online model serving. The offline store is usually a high-throughput, high-latency storage systems like cloud object stores or data lakes. The online store, however, must serve features with tight latency constraints (on the order of 100s of milliseconds). As a result, a smaller subset of features are often stored in in-memory K/V stores (e.g. Redis [7]).

One challenge with splitting feature storage between separate online and offline stores is maintaining *offline/online consistency*. Differences in the ways that features are ingested, materialized, or defined between the offline and online store results in slightly different sets of features being served to training pipelines and inference pipelines. Slight differences in features can results in *data drift* for models, which can cause significant depredations in prediction accuracy. As a result, some feature stores will only allow direct updates to either the offline or online store, and syncs values from one store to another. While this approach can reduce the risk of data drift, synchronization can incur additional overhead and latency in updating features. In this paper, we only focused on materialization for the online store (as we focused on prediction serving), however future work should explore how scheduling policies could affect online and offline consistency.

7.3 Limitations of Existing Feature Stores

Despite being designed for machine learning workloads, existing feature store systems do not account for accuracy in how they maintain feature values. Feature store are uniquely situated between updates to features and feature queried, but typically lack awareness of downstream query patterns and performance of the predictions made using queried features. As a result, systems for maintaining features treat all data updates and keys symmetrically and fail to leverage important information about which updates are critical and which keys are likely to be accessed in the downstream prediction workload. In the online setting, these systems make only a best-effort attempt at maintaining feature values up-to date. Features might become arbitrary stale, significantly hurting accuracy. While the cost of computing features in the online setting excludes keeping features, fully up-to-date, we find the current approach suboptimal.

8 CONCLUSION

In this paper, we studied the challenge of feature maintenance in feature stores, an emerging new class of systems. First, we identified a critical limitation in existing approaches to feature store design: current feature stores treat data and keys symmetrically and do not leverage crucial signal about query access patterns or the impact of features on downstream task performance. We then formalized the feature store problem and introduced *feature store regret*, a metric that measures the impact of staleness of features on the downstream prediction accuracy. Finally, we presented RALF, a feature store system that uses prediction loss as feedback to prioritize updates that improve the downstream accuracy via Regret-Proportional scheduling. Experiments on a range of feature maintenance policies demonstrate that prioritizing replacing features with the highest *cumulative regret* can significantly improve prediction accuracy in resource-constrained settings. We believe this paper will provide a formal foundation for a key problem in the emerging class of feature store systems and hope that it will inspire future work in the design of more advanced feature maintenance strategies.

ACKNOWLEDGMENTS

This work was supported by gifts from AMD, Anyscale, Astronomer, Google, IBM, Intel, Lacework, Microsoft, Samsung SDS, Uber, and VMware. We also thank Abhinav Mishra and Ram Sriharsha for help with understanding feature store challenges at Splunk.

REFERENCES

[1] Amazon Web Services [n.d.]. *Amazon SageMaker Feature Store*. Amazon Web Services. Retrieved November 11, 2023 from https://aws.amazon.com/sagemaker/feature-store/

[2] Hopsworks [n.d.]. *Feature Stores Org*. Hopsworks. Retrieved November 11, 2023 from https://www.featurestore.org/

[3] [n.d.]. *Feature Tools*. Retrieved November 11, 2023 from https://featuretools.alteryx.com/en/stable/getting_started/handling_time.html

[4] Hopsworks [n.d.]. *Hopsworks*. Hopsworks. Retrieved November 11, 2023 from https://www.hopsworks.ai/

[5] grouplens [n.d.]. *MovieLens 1M Dataset*. grouplens. Retrieved November 11, 2023 from https://grouplens.org/datasets/movielens/1m/

[6] OpenAI [n.d.]. *Rate Limits*. OpenAI. Retrieved November 11, 2023 from https://platform.openai.com/docs/guides/rate-limits?context=tier-free

[7] [n.d.]. *Redis*. Retrieved November 11, 2023 from https://redis.io/

[8] Cohere [n.d.]. *Scalable, affordable pricing*. Cohere. Retrieved November 11, 2023 from https://cohere.com/pricing

[9] Tecton [n.d.]. *Tecton*. Tecton. Retrieved November 11, 2023 from https://www.tecton.ai/

[10] [n.d.]. *Timely Dataflow*. Retrieved November 11, 2023 from https://timelydataflow.github.iÆŠo/timely-dataflow/

[11] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 265–283.

[12] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*. 29–42.

[13] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination Avoidance in Database Systems. *Proc. VLDB Endow.* 8, 3 (nov 2014), 185–196. https://doi.org/10.14778/2735508.2735509

[14] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. 2012. Probabilistically Bounded Staleness for Practical Partial Quorums. *Proc. VLDB Endow.* 5, 8 (apr 2012), 776–787. https://doi.org/10.14778/2212351.2212359

[15] Romil Bhardwaj, Zhengxu Xia, Ganesh Ananthanarayanan, Junchen Jiang, Nikolaos Karianakis, Yuanchao Shu, Kevin Hsieh, Victor Bahl, and Ion Stoica. 2020. Ekya: Continuous Learning of Video Analytics Models on Edge Compute Servers. *arXiv preprint arXiv:2012.10557* (2020).

[16] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. 2017. Approximate Query Processing: No Silver Bullet. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 511–519. https://doi.org/10.1145/3035918.3056097

[17] Rada Chirkova, Jun Yang, et al. 2011. Materialized views. *Foundations and Trends in Databases* 4, 4 (2011), 295–405.

[18] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1277–1288.

[19] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 153–167.

[20] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. 2020. InferLine: latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 477–491.

[21] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A low-latency online prediction serving system. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 613–627.

[22] Natacha Crooks, Youer Pu, Nancy Estrada, Trinabh Gupta, Lorenzo Alvisi, and Allen Clement. 2016. TARDiS: A Branch-and-Merge Approach To Weak Consistency. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1615–1628. https://doi.org/10.1145/2882903.2882951

[23] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. 2014. Exploiting Bounded Staleness to Speed up Big Data Analytics. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, USA, 37–48.

[24] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M Frans Kaashoek, and Robert Morris. 2018. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 213–231.

[25] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Mingwei Chang. 2020. Retrieval augmented language model pre-training. In *International conference on machine learning*. PMLR, 3929–3938.

[26] Brandon Holt, James Bornholt, Irene Zhang, Dan Ports, Mark Oskin, and Luis Ceze. 2016. Disciplined Inconsistency with Consistency Types. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (Santa Clara, CA, USA) *(SoCC '16)*. Association for Computing Machinery, New York, NY, USA, 279–293. https://doi.org/10.1145/2987550.2987559

[27] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. 2018. Chameleon: scalable adaptation of video analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 253–266.

[28] Theofilos Kakantousis, Antonios Kouzoupis, Fabio Buso, Gautier Berthou, Jim Dowling, and Seif Haridi. 2019. Horizontally scalable ml pipelines with a feature store. In *Proc. 2nd SysML Conf., Palo Alto, USA*.

[29] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2017. Noscope: optimizing neural network queries over video at scale. *arXiv preprint arXiv:1703.02529* (2017).

[30] Daniel Kang, Edward Gan, Peter Bailis, Tatsunori Hashimoto, and Matei Zaharia. 2020. Approximate selection with guarantees using proxies. *arXiv preprint arXiv:2004.00827* (2020).

[31] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix factorization techniques for recommender systems. *Computer* 42, 8 (2009), 30–37.

[32] Peter Kraft, Daniel Kang, Deepak Narayanan, Shoumik Palkar, Peter Bailis, and Matei Zaharia. 2019. Willump: A statistically-aware end-to-end optimizer for machine learning inference. *arXiv preprint arXiv:1906.01974* (2019).

[33] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. 2013. MDCC: Multi-Data Center Consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic) *(EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 113–126. https://doi.org/10.1145/2465351.2465363

[34] N Laptev and S Amizadeh. 2015. Yahoo anomaly detection dataset s5. *http://webscope.sandbox.yahoo.com/catalog.php* (2015).

[35] Kenton Lee, Ming-Wei Chang, and Kristina Toutanova. 2019. Latent retrieval for weakly supervised open domain question answering. *arXiv preprint arXiv:1906.00300* (2019).

[36] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[37] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making Geo-Replicated Systems Fast as Possible, Consistent When Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, USA, 265–278.

[38] Mengtian Li, Yu-Xiong Wang, and Deva Ramanan. 2020. Towards streaming perception. In *European Conference on Computer Vision*. Springer, 473–488.

[39] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 401–416. https://doi.org/10.1145/2043556.2043593

[40] Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Timothy Roscoe. 2020. Shared Arrangements: practical inter-query sharing for streaming dataflows. *Proc. VLDB Endow.* 13, 10 (2020), 1793–1806. https://doi.org/10.14778/3401960.3401974

[41] Abhinav Mishra, Ram Sriharsha, and Sichen Zhong. 2021. OnlineSTL: Scaling Time Series Decomposition by 100x. *arXiv e-prints* (2021), arXiv–2107.

[42] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging {AI} applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 561–577.

[43] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *CoRR* abs/1906.00091 (2019). https://arxiv.org/abs/1906.00091

[44] Charles Packer, Vivian Fang, Shishir G Patil, Kevin Lin, Sarah Wooders, and Joseph E Gonzalez. 2023. MemGPT: Towards LLMs as Operating Systems. *arXiv preprint arXiv:2310.08560* (2023).

[45] Skipper Seabold and Josef Perktold. 2010. statsmodels: Econometric and statistical modeling with python. In *9th Python in Science Conference*.

[46] Swaminathan Sivasubramanian. [n.d.]. Amazon dynamoDB: a seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD*

International Conference on Management of Data. 729–730 year=2012.
[47] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. 2013. Consistency-Based Service Level Agreements for Cloud Storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 309–324. https://doi.org/10.1145/2517349.2522731
[48] Zhaohui Wang, Xiao Lin, Abhinav Mishra, and Ram Sriharsha. 2021. Online Changepoint Detection on a Budget. In *2021 International Conference on Data Mining Workshops (ICDMW)*. IEEE, 414–420.
[49] M. H. Wong and D. Agrawal. 1992. Tolerating Bounded Inconsistency for Increasing Concurrency in Database Systems. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (San Diego, California, USA) *(PODS '92)*. Association for Computing Machinery, New York, NY, USA, 236–245. https://doi.org/10.1145/137097.137880
[50] Kun-Lung Wu, P.S. Yu, and C. Pu. 1992. Divergence control for epsilon-serializability. In *[1992] Eighth International Conference on Data Engineering*. 506–515. https://doi.org/10.1109/ICDE.1992.213158
[51] Haifeng Yu and Amin Vahdat. 2000. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4* (San Diego, California) *(OSDI'00)*. USENIX Association, USA, Article 21.
[52] Jingren Zhou, Per-ke Larson, and Jonathan Goldstein. 2005. Partially materialized views. In *submitted to this conference*.