Automated Cache Hierarchy for Feature Stores

Priyam Mohanty, Shreyas Krishnaswamy, Edward Choi
University of California, Berkeley
{priyam.mohanty,shrekris,edwardc1028}@berkeley.edu

Abstract

Real-world data pipelines are becoming more and more reliant on data featurization. This has propelled development of feature stores, systems that store featurized data and serve them to ML models or other workloads downstream. While many production feature stores are ad-hoc, a recently-developed, unified feature store called RALF has facilitated research on end-to-end feature store optimizations. However, RALF currently assumes all features fit in memory. This inhibits its ability to handle large workloads. Thus, a caching hierarchical model is necessary to evict features and save to disk when necessary. Towards this model, we introduce three new novel caching policy "classes" specific to feature stores: cost-awareness, table-level caching, and a combination of the two. For rapid protoyping, we design and build a feature store simulator that implements these policy classes and demonstrate that these policy classes perform considerably better than naive caching strategies like LRU and MRU. We also perform a validation experiment to confirm that the simulator actually mirrors RALF in terms of functionality and speed.

1. Introduction

Machine learning models have made a roaring comeback in hundreds of different industries today. As data continues to be abundant and easier to collect, machine learning prominence will continue to grow in the real-world. As a result, an entire industry has risen to create new tools to help data scientists and machine learning engineers. Data featurization is an integral part of the data science lifecycle. A pressing challenge nowadays is ability to store features, especially complex ones that take tremendous compute power or are derived from multiple data sources, to avoid unnecessary recomputation. One solution is the use of feature stores. These systems store features derived from raw data and serve them to downstream models for training and prediction.

Modern machine learning pipelines constantly need to adjust to real-time data, such as a recommendation model of trending content or estimated wait time for a restaurant order. Feature stores are useful here because they allow for shared computation and optimizations. Various models can access a centralized feature store which allows for easy feature access across various teams and use cases, both for training and inference. Feature stores are meant to keep features composable and extensible, while also keeping features up to date with real-time data if needed.

Today's feature stores are built atop various technologies. In this paper, we focus on RALF (Realtime, Accuracy & Latency-aware Featurization) [5, 20], a key-value store written in Python and built on top of Ray [15]. Ray provides a Python API for building distributed applications. RALF uses a key-value store consisting of "Table" objects that store data records.

RALF, in its current state, is extremely fast as a feature store, but lacks efficient caching capabilities. RALF leverages Ray's object spilling to handle larger-than-memory workloads using disk [19]. However, Ray spills to disk using a naive policy: when memory is full, then objects start being spilled to disk. Thus, there is room for caching optimizations that take advantage of unique feature store properties and access patterns.

To summarize, we make the following contributions:

- Three novel **caching policy "classes"** that leverage unique feature store traits to enhance existing, naive caching strategies
- Three novel **workloads** that exemplify traditional feature store use-cases and their impact on memory
- An extensible and general-purpose **feature store simulator** that enables rapid caching prototyping for feature stores

2. Related Work

As the goal for our paper is to build a caching system for feature stores, this section covers two important parts: existing feature store solutions and caching policies. By analyzing both in more detail, we gather information about feature store-specific qualities and also explore how we can develop

caching policies related to them. We note that feature stores are still a relatively new concept, so there has been limited research literature on the subject, but there have been many widescale open-source and enterprise projects.

2.1. Feature Stores

Aside from RALF, there are many feature stores available as open-source and enterprise projects.

Michelangelo. Uber's Michelangelo machine learning platform [11] paved the way for handling and scaling machine learning workloads. Its goal was to be an ML-as-a-service platform which automated different aspects of the machine learning model lifecycle, which includes: managing data, training, evaluating, and deploying models, and making and monitoring predictions. While the idea of a scalable machine platform was not novel at the time in 2017 [14], Uber was the first to introduce the notion of feature stores, which would allow engineering teams to share, discover, and curate a set of features for their machine learning problems. In essence, this was the birth of an integrated feature management solution and has since led to a wide array of new solutions, including RALF.

Doordash. DoorDash's feature store [12] is built on top of Redis, but they encountered many problems with inefficiency and insufficient memory capacity. Thus, they sought to perform several optimizations to boost read throughput as they needed to perform several millions of feature lookups per second. For example, they were able to reduce memory footprint by using string hashes on feature names, protocol buffers [4] for vector embeddings/integer lists, and Snappy compression [6] for list compressions. Through these optimizations, DoorDash was able to improve their read latency from Redis by 40% and their overall feature store latency by 15%. DoorDash's optimization methodology shows that exploiting knowledge of the internals of the feature store allows for possible optimization, which is an idea we used when devising our caching policies, such as cost-aware and table-level caching.

Feast. Feast [2] is an open-source feature store that provides the ability to store and server features for model inference and training. A big limitation to the system unlike RALF, for example, is that Feast assumes feature transformations are done separately, which means users must provide external systems that perform feature computations.

Tecton. Tecton was founded by the team that created the Uber Michelangelo platform with the goal to provide enterprise-ready feature store and make machine learning accessible to all companies. Tecton, unlike Feast, is actually able to perform feature computations as part of the internal feature store system. The Tecton feature store platform also consists of several components, including the feature registry, serving, storage, transformations, and monitoring. [7] Most of these components exist in RALF but it is currently lacking monitoring which is meant to validate data for correctness and detect data drift.

Amazon SageMaker. Amazon's SageMaker feature store [1] exists as part of the Amazon AWS ecosystem. SageMaker is able to ingest data from many sources and interactive queryability of features through Amazon Athena. However, a big limitation of Amazon SageMaker is its reliance on the AWS ecosystem, making external integrations much more challenging.

2.2. Caching Policies

While, to our knowledge, there are no existing feature store-specific caching policies, we can draw inspiration from caching policies used in other domains, such as web applications, to devise our caching schemes.

Hyperbolic caching [9] is a caching policy for web applications which is meant to tailor caching strategies specific to the needs of the application (i.e. by including fetching cost or expiration time). At the most basic level, hyperbolic caching employs the following priority calculation formula:

$$p_i = \frac{n_i}{t_i}$$

where the priority p_i for an item i in the cache is defined by n_i which represents the request count for i since it entered the cache and t_i which represents the time since it entered the cache. Thus, hyperbolic caching allows a new item's priority to converge to its true popularity from an initially high estimate, when over time the priority of the item will drop along a hyperbolic curve. Hyperbolic caching is also extensible because the priority formula can be easily adjusted to be expiration-aware or cost-aware.

The ability for hyperbolic caching to be expiration-aware is useful because feature stores must consider feature freshness, especially in a real-time-context, and extending the caching policy by making it expiration-aware allows the policy to evict stale features easily.

Cost-awareness is also relevant in our paper because in the feature store, we can factor in features' computational and disk costs and allow hyperbolic caching to change the priority of items accordingly. Cost-awareness has also been implemented in other ways such as the GreedyDual algorithm [21], a primal-dual strategy for solving the problem of making replacement decisions for items with non-uniform costs, or an amortized constant-time implementation of this algorithm, known as *GD-Wheel* [13].

3. Metrics of Success

The goal of this project is to devise and evaluate caching policies that leverage feature store specific attributes. Thus, our three metrics of success are:

- **Minimize query latency.** Our primary metric of success is minimizing query latency. We define query latency as the end-to-end time [17] it takes to fetch a feature whether it is in the cache, on disk, or computed on-the-fly. We will primarily experiment with 4 caching techniques: LRU, MRU, Hyperbolic, and Random. These will be fitted with two different types of optimizations: cost-aware and table-level caching. Cost-aware caching will only evict a record from the cache if it's cheaper to recompute it than it is to save and fetch it from disk. Table-level caching involves exploiting a specific attribute of feature stores such that we can prefetch or pre-prioritize records within individual tables that are accessed frequently to take advantage of table locality.
- **Beat any and all naive caching policies.** A naive caching policy is one that does not account for feature store-specific optimizations, such as any application-agnostic caching strategy. We test our policies with LRU, MRU, Hyperbolic, and Random.
- **Get as close to memory-only performance as possible.** Memory-only performance is impossible for workloads larger than memory, but it provides a useful lower-bound on caching performance. We aim to create optimizations that get as close to memory-only performance as possible.

4. Design Overview

4.1. Feature Store Simulator

4.1.1 Motivation

As the RALF codebase [5] was complex to navigate and implementing our various caching policies (i.e. Hyperbolic Caching) required breaking through several layers of abstraction, we opted to build a feature store simulator in order to iterate faster. This simulator drastically cut down on our development and iteration time and the results gathered from it were able to reflect the actual feature store implementation, RALF. To prove this, we validate that our simulator implementation and RALF align in section 4.1.7.

4.1.2 Tables

Like in RALF, out simulator also has the concept of tables which keep track of records, but unlike real tables in RALF which encapsulate an "operation" that is applied to incoming records, simulated tables instead track an "operation latency", which is the expected time it would take to apply the operation to a record. Each table has a simple interface to insert records and query records based on a key. A table also maintains a list of "upstream tables" which are references to other tables that contain records which the current table's records depend on. An example of upstream dependencies can be seen in Figure 1.

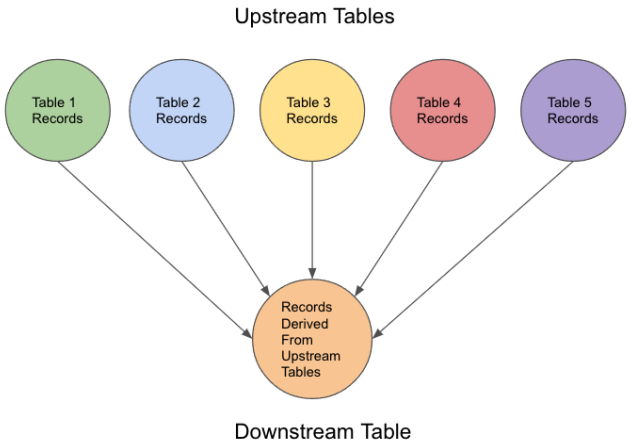

Figure 1. Downstream table records are derived from upstream tables' records.

4.1.3 Records

Like in RALF, our simulator has the concept of records. We note that we use the words "records" and "features" interchangeably throughout the paper. Records are composed as key/value pairs where the key is a hashable object type, but unlike in RALF where record values hold the actual data, our simulator holds an intrinsic value of the record through the composition of 3 elements:

1. **Record size in number of bytes**: Represents the record's size in number of bytes.
2. **Concatenated version numbers of immediate upstream dependency records**: As a record can have multiple upstream dependencies, we keep track of each upstream dependencies' version number by concatenating them together. A record's upstream dependencies are all the records in upstream tables used in calculating that record. We denote records that do not have upstream dependencies as raw features.
3. **Record's version number**: The current record's version number which may be used for an upstream dependency for a later downstream dependency.

4.1.4 Version Maintenance

A major distinction of our simulator from RALF is that features are derived from upstream dependencies rather than downstream dependencies. We decided to use this model of upstream dependencies because only when a record is queried do the record's values and upstream records get

updated. This allows us to execute queries and lazily update upstream dependencies only when necessary as compared to downstream-based models where upstream records constantly propagate to downstream records which is more computationally intensive.

To check whether an upstream record has been changed, the downstream table can compare the upstream record's version with its cached version in the downstream record's version string. If it has changed, it needs to be recalculated, and then its version number is incremented by 1. If an upstream record hasn't been updated, then downstream records that depend on it don't have to wait for it to be rematerialized. The query latency for that upstream record is set to zero.

4.1.5 Memory Manager

The Memory Manager tracks the amount of memory taken up by records inside of Tables. Record accesses must go through the Memory Manager to simulate memory accesses and disk accesses. The Memory Manager is composed internally of a memory cache, where its size is determined during the memory manager instantiation, and disk, which is simulated as a dictionary. To ensure that the latencies from reading/writing to the cache and disk in the memory manager reflect numbers used in real-life, we scaled our numbers accordingly [3, 18].

Another important function the memory manager has is to handle evictions. During instantiation, the memory manager must also take in an eviction policy, which is defined by a set of eviction policies that we discuss in more detail in section 4.2.

To facilitate the processes outlined above that occur in the memory manager, we define a simple `get` and `set` interface:

- `set(key, record)`: The responsibility of the `set` method is to set a record in the memory manager. As our memory manager uses write-back caching, a record that needs to be placed in the memory manager is only initially placed in the memory cache, after which the eviction policy will determine the eviction candidates. When candidates are evicted, the records will then be saved to disk. An optimization we made during the eviction process is that if the record is not a raw feature and the associated operator latency for the record $<$ (disk read + write latency), then it is more optimal to recalculate the record on-the-fly rather than incur the more expensive cost of disk reading and writing. We must save raw features to disk because there is no chance of recalculating them once they are evicted from the memory cache.

- `get(key)`: The responsibility of the `get` method is to retrieve a record by key from the memory manager. If the record exists in the memory cache, then the record is returned from the memory manager and the computation is finished. However, if the record does not exist, then a disk fetch is necessary. Once the record is fetched from disk, it is saved to the memory cache and eviction candidates are determined and evicted based on the eviction policy. One caveat for the fetch from disk is that the record might not actually exist, because of our cost-aware optimization described above. To resolve this issue, it is up to the caller of the memory manager (i.e. the Tables) to recalculate the record. For example the calculation is performed in the Table's `query` method.

4.1.6 Limitations

Our simulator has three limitations:

1. **Single-threaded** : The current system currently assumes that all code runs in a single-threaded fashion, and does not make use of concurrency from libraries such as Ray and asyncio which are used extensively in RALF. We opted to keep our system single-threaded to simplify debugging and reduce the overall complexity especially when dealing with caching and evictions in a multi-threaded context.

2. **Timestamps**: The current implementation has no notion of time. Time is ill-defined in the simulator because there's no concurrency, so although different queries could execute concurrently in reality, the simulator can't track this. Instead, we use the concept of timesteps measured in seconds in our results.

3. **Fragmentation**: The current system assumes fragmentation does not exist. It doesn't map records to specific places in memory. Instead, the simulator just deals with total memory used.

4.1.7 Validation

In order to confirm that our simulator mirrors the performance and functionalities of RALF, we performed a validation experiment. As we can see in Figure 2, our validation involved creating a chain of operations in both the simulator and RALF. Each circle represents a table and has some operator attached to it. The source table operator generates records and adds them to the memory manager. Records belonging to the Long Latency Operator are written to disk while records belonging to the Short Latency Operator are only written to disk without the cost-aware optimization.

As RALF is downstream-based while the simulator is upstream-based, insertions in the simulator and queries in RALF have a similar impact on memory, as do queries in the

	RALF (sec)	Simulator (sec)
Optimized (O)	0.066	0.711
Non-Optimized (NO)	0.165	1.889
Ratio (O/NO)	2.486	2.658

Table 1. This table contains simulator validation results. Both RALF and the simulator ran a valdiation workload using naive LRU and cost-aware LRU, and the mean latencies are displayed in seconds. The mean latency ratio's approximate difference of only roughly 7% provides evidence of the simulator's accuracy.

simulator and insertions in RALF. By validating the simulator on a sequential feature store, the impact of caching can be studied by comparing the queries in the simulator with the insertions in RALF (and vice versa) since both pairs of actions take the same path in memory. In order to run this workload, RALF was equipped with a custom application-level memory manager that enforced carefully timed disk reads and disk writes upon eviction. The application-level memory manager was used in tandem with operators that would access the disk whenever requested to do so by the memory manager, allowing the feature store to measure latency end-to-end. RALF was also tuned to perform disk accesses that took roughly the same amount of time as the simulated accesses from the simulator.

In the simulator, the source table takes in record insertions. However, at this stage, the source table records are not immediately propagated downstream to the short latency operation and long latency operation tables, as the downstream output table has not queried from these 2 tables yet; however, the 2 tables are aware of their upstream tables' existence. After the output table queries for records, the upstream table operations are then performed and any calculations that need to be done for the records are performed and propagated from the intermediate short latency operation and long latency operation tables down to the output table.

In RALF's validation layout, the source table generates records which propagate through the operators until they reaches the writer table operator, which writes the amount of time taken to compute individual feature records from end-to-end (Source Table $\rightarrow$ Long Latency Operation $\rightarrow$ Short Latency Operation $\rightarrow$ Output Table).

The validation workload consists of 100 records that were inserted into the feature store. The records were accessed 1000 times according to a random Gaussian distribution. The feature store used an LRU caching strategy under both the naive and the cost-aware policy classes. As seen in Table 1, RALF sees a 2.658x speedup with the optimization while the simulator sees a 2.486x speedup (roughly a 7% difference). The difference likely arose from the limitations of the simulator, including primarily a lack of concurrency.

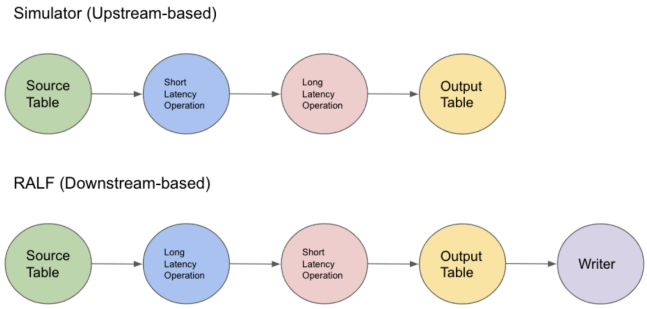

Figure 2. Simulator vs. RALF Validation

4.2. Caching Policies

Since feature stores generally glue together disparate resources, such as streaming, batch, caching, and storage systems [20], they usually rely on the caching policies of the underlying architecture. Each of these resources often have their own caching policies which in turn lack visibility into other caching policies. Tuning and coordinating these policies requires burdensome engineering efforts across multiple third-party tech stacks. As a result, industry efforts have instead focused on higher abstraction levels such as resource selection and feature management [8, 12]. However, RALF's unified nature opens the possibility of end-to-end caching policies that can leverage the connectivity between different parts of the feature store when making caching decisions. Such decisions can also leverage unique feature store properties.

Generally, existing caching policies take advantage of access patterns heuristics such as temporal locality when deciding which pages to evict or keep in memory. These policies, such as LRU or MRU, remain broadly useful for feature stores because the same general principles apply. Temporal locality, for instance, still applies to feature stores – in many workloads, data records that are accessed are likely to be accessed again soon. Given this observation, much of the policy exploration in this paper focuses on developing policy "classes" that are well-suited for feature stores specifically. These policy classes focus on delivering feature store-wide optimizations and can be attached to more common eviction policies such as LRU. Such an abstraction pattern provides the best of both worlds: feature store operators can improve performance using intuitive and commonly-known policies well-suited to their particular workloads while also getting feature store-specific modifications that improve the naive caching policies.

4.2.1 Naive Policies

The naive policies that are further optimized in this paper include:

- **Least Recently Used (LRU)**: This policy is standard LRU implemented with a min-heap.
- **Most Recently Used (MRU)**: This policy is standard MRU implemented with a min-heap.
- **Random**: This policy chooses records randomly to evict from the cache.
- **Hyperbolic Caching**: This policy tracks a function composed of a record's cumulative accesses and its time in memory. It guards against bursty workloads, which tend to elevate record frequencies to high levels in short periods of time. This calcifies stale records in memory under purely frequency-based eviction schemes since their high frequencies shield them from eviction. Gradually reducing priority over time ensures that once the bursty workload ends, its records will face priority decay and eventually get evicted.

4.2.2 Feature Store Observations

The feature store simulator's table-based model allows end-to-end caching optimizations that take advantage of operator knowledge about table operations. This paper explores three key observations about feature stores specifically. First, cost-awareness. Since feature stores rely on well-defined operations that are chained together and encapsulated in tables, feature store users can determine table latencies intuitively, empirically, or even heuristically by defining algorithms that learn the latencies by collecting data on tables in an active feature store. This can be coupled with structural details about a particular feature store setup, such as proximity in the store to a queryable table, to make more informed eviction decisions.

Second, access ripples. In a unified feature store, the table network generates an implicit secondary access pattern on top of the user's own access pattern. Since tables are connected in a fixed directed acyclic graph (DAG), insertions and queries in the store cause "rippled" accesses as downstream and upstream tables are queried and updated in response to the user's original request. Since these access ripples touch records across different tables in the feature store, a single user request can have a widespread effect on the memory manager.

Third, table locality. Feature stores can be designed to promote access patterns that repeatedly touch a small radius of tables. As an example, consider a per-user feature store with a single, queryable layer of tables at the end of the store. The tables consist of per-user data where every user gets their own table with data specifically prepared for that user. In such a design, whenever a user accesses the application, they'll make frequent requests to only their own table. More generally, organizing feature stores to promote table locality allows optimizations to leverage these access patterns by prefetching or pre-prioritizing data that will likely be accessed soon.

4.2.3 Cost-Aware Caching

Storing features on disk requires at least one disk write and one disk read for such features to become useful once again. By gauging the compute cost to recalculate a feature, the feature store can instead decide whether to persist the feature in the first place. If it doesn't persist the feature, then when the feature is requested once again in the future, the store needs to recalculate its value from upstream dependencies and reproduce the feature. This strategy helps bypass disk for operations (e.g. averages, sums, etc.) that are quick to complete, and rematerialization in general has proven successful in recent dataflow systems [10]. However, this optimization comes with certain drawbacks.

In particular, time-aggregating features may suffer accuracy loss. For instance, consider a feature store with a table, `request_stats`, that tracks average time between user requests for certain features. Each time a user makes a request that crosses the `request_stats` table, the table can use the current request's timestamp and the previous request's timestamp to modify a running average of the request intervals. The arithmetic operations (addition, subtraction, and averaging) that this table performs have low latency, which makes it a strong candidate for recalculating records rather than saving them. However, a recalculated value will lack historical data, degrading accuracy. A straightforward workaround is disabling this optimization for tables that track historical data; however, future work may also bound the tolerable accuracy loss to determine whether a feature can be evicted without persistence.

However, cost-aware caching allows low-latency tables to operate without relying on disk. Furthermore, cost-aware caching is well-suited to feature stores because these systems rely on real-time data. Raw features that are frequently updated or collected induce staleness in downstream features that depend upon them. Even worse, these real time updates can invalidate downstream features which provide an expired view over them. Forcing downstream tables to persist data even when some upstream features frequently change effectively makes disk accesses an expensive no-op. Cost-aware caching eliminates these no-ops and improves performance (see the "Results" in section 6 for more information).

4.2.4 Table-Level Caching

A feature store's connectivity propagates access across different tables. While the user may be making frequent requests to only a limited number of tables, these table's dependencies will also be queried, causing a propagation ef-

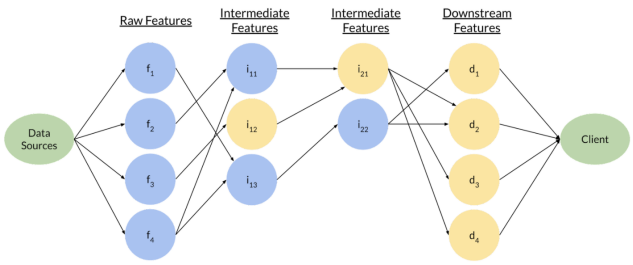

Figure 3. This is a sample feature store layout. The circles represent tables that receive records from upstream dependencies (tables connected to their left) and send records to downstream dependencies (tables connected to their right). The light blue circles are tables that cannot be queried; whereas, the light yellow circles are tables that can be queried. Notice that the intermediate tables i_{12} and i_{21} can both be queried by a client despite no direct arrow between them and the client in the diagram. In a generic feature store, these circles may instead represent operators in entirely disparate systems (e.g. streaming systems like Flink or databases like RocksDB). In a unified store like RALF, however, these circles are each independent tables that remain in a single system. This unification provides tables awareness of one another, undivided by system boundaries that may impede ad-hoc feature stores.

fect on the tables that are touched. Such a propagation will also trigger widespread changes in the caching layer.

For example, in Figure 3, a client querying downstream feature d_1 will have a limited effect on the caching layers since d_1 has limited dependencies: i_{22}, i_{13}, f_1, and f_4. However, a client querying downstream feature d_2 will have an extraordinary effect on the caching layer since every table (except the other downstream tables) is then accessed.

Recall that feature stores can be organized to provide table locality where access patterns touch a small radius of tables frequently. Therefore, the feature store can anticipate which tables will likely be accessed based on accesses to a relatively small group of tables. This motivates table-level caching, which prefetches or pre-prioritizes all records within frequently-accessed tables to take advantage of table locality. Addiionally, the memory manager can prefetch/pre-prioritize upstream dependencies for these tables as well, allowing records to be quickly accessed and remain fresh.

Another intuitive motivation for this table-level caching is to consider what happens when some upstream dependencies are persisted on disk while others are kept in memory. When a downstream record is accessed, it must confirm that its upstream dependencies are up to date; if they are not, the downstream record must be recalculated with fresh upstream data. If even some upstream dependencies are persisted to disk, then although the downstream record may be kept in memory, accessing it still requires accessing disk. In other words, a true memory access for a record also requires its upstream dependencies to remain in memory.

4.2.5 Lazy Evaluation

One other caching pseudo-optimization follows directly from the simulator design. Unlike RALF, the simulator is an upstream-based system. In other words, it lazily evaluates downstream features when requested instead of eagerly evaluating all downstream features as soon as a raw feature is updated. Eager evaluation has detrimental effects on caching because insertions flood the cache with all the records used to update downstream features, regardless of their usage. Under workloads with frequently updated raw features, the cache will frequently be flooded with new records making it difficult to reason about and build reliable caching optimizations.

However, one drawback of lazy evaluation is that downstream features are computed at query time, rather than at insertion time, causing increased latency for the client when querying records. This "cold-start" effect is somewhat mitigated by the other optimizations (cost-awareness and table-level caching). It's also important to note that the decision between lazy evaluation and eager evaluation is not necessarily binary. For instance, one possible improvement to the table-level caching policy is to not only pre-prioritize records but also eagerly evaluate records that may soon be queried. That way, a cold start would be noticed when accessing a table for the first time, but afterwards, that table's records should be quick to access. See "Future Work" in Section 7 for more information.

4.2.6 Caching Policy Classes

This paper proposes the following three novel caching policy classes based on the prior subsections' observations and optimizations:

- **Cost-Aware Caching**: this policy class requires the feature store user to determine a "table latency." The table latency is a float indicating the average time it takes for the table to calculate a new record value given upstream values. When a record must be evicted, the memory manager checks table latency $<$ disk read + write latency). If the table latency is lower, the memory manager will evict the record without persisting it, and when the record is accessed again, the record will be recalculated and then stored in memory.

- **Table-Level Caching**: this policy class updates all record priorities in a table when any record in that table is accessed. For example, under an LRU policy, this policy class will update the priorities for all records in an accessed table to be the most recently used. Since the feature store implicitly accesses upstream tables as

well, this implementation also implicitly updates all upstream table record priorities for an accessed table, taking advantage of the access ripple pattern described in section 4.2.2. It's important to note that this paper's implementation of table-level caching only updates all corresponding table records that are already in memory. It doesn't eagerly prefetch additional records into memory to avoid burdensome disk read/write costs in the single-threaded simulator.

- **Cost-Aware and Table-Level Caching**: This policy class uses both cost-aware and table-level caching simultaneously. It is particularly well-suited to feature store layouts with low-latency tables and access patterns that express table locality. However, its performance can suffer when either of these conditions are not met since the corresponding optimizations become detrimental to the feature store's performance.

5. Workloads

We use novel workloads to test our caching policies.

First, a Correctness workload verifies that the caching system works. This workload "stress tests" the system and simulator by spawning roughly 10,000 unconnected tables and executing tens of thousands of queries on them.

Then, three novel workloads were developed to actually evaluate caching strategies.

1. **Real-Time Predictions** : This is a generic workload based on a real-life, real-time workload at a companies like DoorDash or Uber.

2. **Trending Recommendations**: This is a variation of the above workload minus a "real-time" aspect, so this workload represents features that need not be real-time or updated constantly.

3. **Fraud Detection**: - This workload has an access pattern that emphasizes table-level locality: the same tables are accessed multiple times in short bursts.

5.1. Correctness Workload

The correctness workload contains independent tables with arbitrary numbers of records. The tables are unconnected. Additionally, tables are not queried multiple times, rendering caching ineffective. This test gauges simulator correctness by confirming that caching optimizations have negligible effect. However, since this workload produces no interesting or insightful results, its data is excluded from this paper.

5.2. Real-Time Predictions Workload

The Real-Time Predictions workload replicates a real-life, real-time workload at companies like Doordash [12]. The model workload estimates the average time needed to arrive at a location based on a moving average of the past five days. While the simulated logic simplifies the real process's inner details, it accurately preserves the workload's effects on the cache. The workload generates 14 tables with each table's key corresponding to a date in the last 14 days. Then, records of "trip data" are inserted into each table, including details such as order time, delivery time, and more. Lastly, each table depends on (up to) the previous five days of data, which means that each table has up to five upstream tables. Then, all tables are queried sequentially from the beginning of the month to the end, simulating the workload's real-time nature of real-time, where anything earlier than the last 5 days of a query are never again accessed again.

5.3. Trending Recommendations Workload

Our Trending Recommendations workload is similar to the above workload, but it lacks the "real-time" aspect. In other words, all 14 days of data within the table may be queried in the workload at any point, unlike the linear accesses of the previous workload. This workload's model is "trending recommendations," such as Netflix's recommendations system which is updated offline and only periodically [8]. Like before, the workload creates 14 tables, each corresponding to 14 dates. Each table also gets data records and an arbitrary number of upstream tables. To choose arbitrarily, a random integer between 1 to 5 determines the number of previous upstream tables for the current table. Lastly, these tables are accessed in a random order.

5.4. Fraud Detection Workload

The fraud detection workload's main goal is simulating table locality by accessing the same table multiple times in a row. This is a common business use case where a fraud detection algorithm may need to access a user's table or a transaction table repeatedly in real-time to make risk assessments. Low-latency, real-time machine learning has delivered greater utility and business value compared to traditional use cases like post-hoc analytics [16]. Towards this aim, this workload creates multiple tables and fills them up with data, like in the previous workload examples. However, the query pattern differs. Each table is queried 5 times before the workload moves to the next table access.

6. Evaluation

Our experiments were run on a single AWS EC2 instance running Ubuntu 20.04 LTS on an Intel Xeon CPU E5-2676 v3 @ 2.40GHz. The instance was provisioned with 8 GiB of RAM and 128 GiB Standard SSD GP2 storage.

Using the three workloads described above, we tested our caching policies, combined with a caching strategy (LRU, MRU, Hyperbolic, Random), for each one. Each

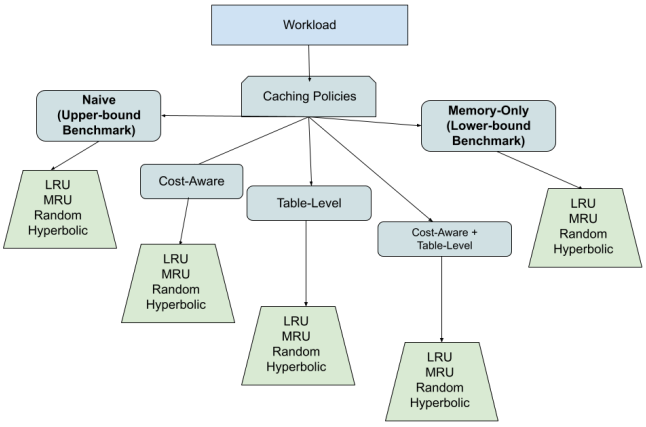

Figure 4. Each workload was run with 20 combinations. Each of the five caching policies was run with 4 caching strategies, for a total of 20. The Naive caching policy served as an upper-bound benchmark (always want to beat), while the Memory-only caching policy served as a lower-bound benchmark (get as close to as possible).

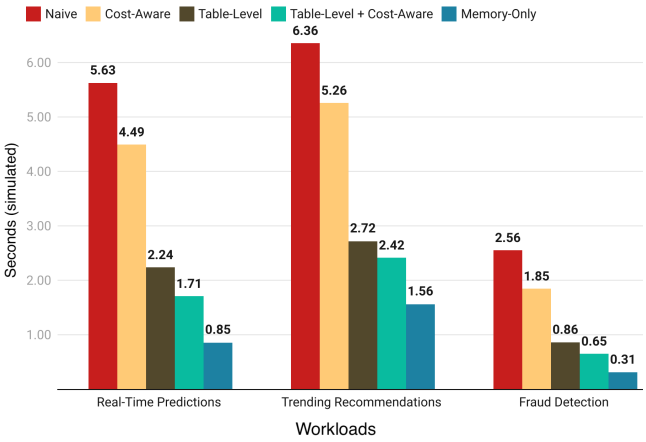

Figure 5. **Average R/W Latency for Workloads Per Caching Policy.** Naive (red) is the upper-bound benchmark and Memory-Only (blue) is the lower-bound benchmark. The Cost-Aware + Table-Level caching policy performs best across all workloads.

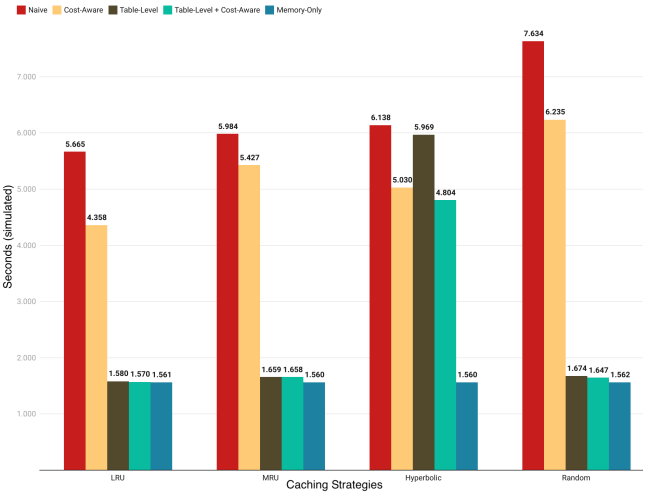

Figure 6. **Trending Recommendations Workload Caching Performance.** Naive (red) is the upper-bound benchmark and Memory-Only (blue) is the lower-bound benchmark.

workload was run with a total of 20 specific caching policy-strategy combinations (Figure 4). For each workload, we have two benchmarks to beat: Naive and Memory-Only. The Naive pass had none of our caching policies applied, while the Memory-Only pass was given an extremely high level of memory so as all objects were able to fit in memory.

The Naive benchmark is the performance of a standard caching strategy (LRU, MRU, Hyperbolic, Random) without any specific optimizations applied. This is the default performance of RALF currently when spilling to disk. It is the upper bound of our metrics of success, and the goal is to always beat this.

The Memory-Only benchmark measures the performance of our workloads if all the data within the workloads fit into memory. This means that there is no writing to disk and as a result, no caching implementation needed. It is the lower bound of our metrics of success: we want to get as close to this number as possible.

6.1. Evaluating Caching Policy Classes

In our simulator, we found that on average, the caching policy class "Table-Level + Cost-Aware" performed best. It consistently outperformed every other class, including the Naive benchmark, for every workload. However, as we delve deeper into the data and separate each class within each workload, we find that the combination of LRU and "Table-Level + Cost-Aware" performs the best for every use case. In general, LRU fits well with most access patterns, especially for feature stores, where specific sets of data may be accessed commonly. However, naive LRU is not always the best option, such as in the Real-Time Predictions workload where it actually performs the worst. With our optimizations, we are able to get LRU to perform nearly as well as our memory-only benchmark, usually within a 5% range. MRU came in at a close second, while hyperbolic and random were too inconsistent to be useful here.

We also notice that hyperbolic caching with a table-level + cost-aware combined caching policy performs significantly worse than its peers in the same policy. This is evident in each workload-specific graph and pulls the average up within Figure 5.

In regards to other policies, we see that our cost-aware policy still delivers performance gains over a naive policy. Generally, table-level policies perform better than cost-aware in almost all cases.

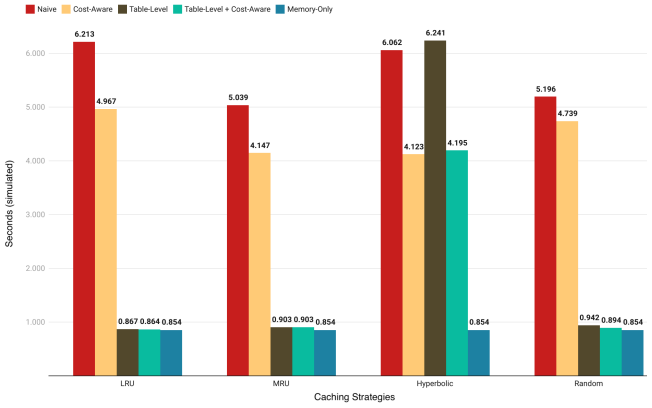

Figure 7. **Real-Time Predictions Workload Caching Performance.** Naive (red) is the upper-bound benchmark and Memory-Only (blue) is the lower-bound benchmark.

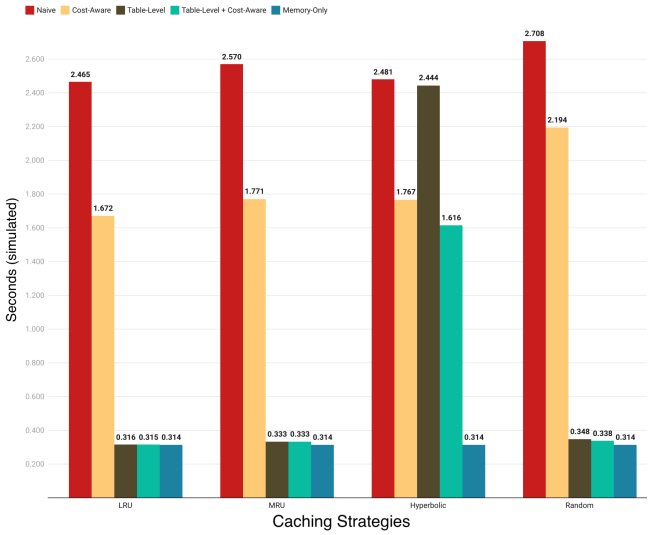

Figure 8. **Fraud Detection Workload Caching Performance.** Naive (red) is the upper-bound benchmark and Memory-Only (blue) is the lower-bound benchmark.

However, we can conclude that our combined cost-aware and table-level caching policy delivers the most significant performance gains to any caching strategies implemented.

7. Future Work

This paper motivates a few areas for future work. First and foremost: accuracy-aware caching. Developing caching strategies that are accuracy-aware could allow for novel caching decisions based on raw feature update frequency as well as staleness tolerance. Hyperbolic caching in particular provides a promising path to implementing an accuracy-based caching system since the accuracy heuristic can substitute previous work that involved cost heuristics. Another path forward could be developing measures of accuracy to allow the feature store user to bound the accuracy degradation that they're willing to tolerate.

Another area of future work is exploring the gradient between eager evaluation and lazy evaluation. This paper's simulator relies on lazy evaluation to allow feature store users and caching policy designers to better reason about the effects of caching on feature stores. Completely eager evaluation can have widespread but unwanted effects on the cache upon inserts. However, completely lazy evaluation means latency is offloaded to query-time, which degrades the user experience. Instead, one possible middle-ground is prefetching with table-level caching. For instance, if a table's record is queried, the feature store can then pre-query records from that table (as well as upstream dependencies), which would allow the cache to take advantage of table-level locality while also removing some query-time latency. Any performance degradation can then be contained in a temporary "cold-start."

8. Conclusion

In this paper, we demonstrate that taking a combined cost-aware and table-level caching approach can lead to significant performance improvements compared to naive caching strategies within feature stores. To expedite iteration, we develop a caching simulator of RALF to discover optimization opportunities. With such optimizations, we hope future feature stores offer reduced latency guarantees.

We also develop workloads motivated by real-world use cases and query patterns to test caching policy classes and caching strategies. On the workloads, we also find that using an LRU caching strategy combined with a cost-aware approach with feature store table optimizations proves to be the most performant caching option explored.

Furthermore, we hope that our simulator proves useful in future work on RALF in order to save iteration time and compute power compared to running a real workload. We look forward to our work being used within RALF to not only develop an automated cache hierarchy, but also to improve upon naive caching policies. While there isn't much research yet on feature store caching, we are excited to see the space gain traction.

9. Acknowledgments

We would like to thank:

- Sarah Wooders for her helpful advice and guidance on feature stores and RALF.
- Stephanie Wang for her detailed guidance in Ray and helping jumpstart this project.
- Professor John Kubiatowicz for providing advice on our feature store validation.

References

[1] Amazon sagemaker feature store. https://aws.amazon.com/sagemaker/feature-store/.

[2] Feast feature store. https://feast.dev/.

[3] Latency numbers every programmer should know. https://colin-scott.github.io/personal_website/research/interactive_latency.html.

[4] Protocol buffers. https://github.com/protocolbuffers/protobuf.

[5] RALF Codebase. https://github.com/feature-store/ralf/.

[6] Snappy compression. https://github.com/google/snappy.

[7] Enterprise feature store for machine learning, Nov 2021. https://www.tecton.ai/.

[8] Xavier Amatriain and Justin Basilico. Netflix Recommendation Service. https://netflixtechblog.com/system-architectures-for-personalization-and-recommendation-e081aa94b5d8/.

[9] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. Hyperbolic caching: Flexible caching for web applications. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, page 499–511, 2017.

[10] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. Noria: Dynamic, partially-stateful data-flow for high-performance web applications. OSDI'18, page 213–231, USA, 2018. USENIX Association.

[11] Jeremy Hermann. Meet michelangelo: Uber's machine learning platform, Mar 2020. https://eng.uber.com/michelangelo-machine-learning-platform/.

[12] Arbaz Khan and Zohaib Sibte Hassan. DoorDash Feature Store. https://doordash.engineering/2020/11/19/building-a-gigascale-ml-feature-store-with-redis/.

[13] Conglong Li and Alan L. Cox. Gd-wheel: A cost-aware replacement policy for key-value stores. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[14] Akshay Naresh Modi, Chiu Yuen Koo, Chuan Yu Foo, Clemens Mewald, Denis M. Baylor, Eric Breck, Heng-Tze Cheng, Jarek Wilkiewicz, Levent Koc, Lukasz Lew, Martin A. Zinkevich, Martin Wicke, Mustafa Ispir, Neoklis Polyzotis, Noah Fiedel, Salem Elie Haykal, Steven Whang, Sudip Roy, Sukriti Ramesh, Vihan Jain, Xin Zhang, and Zakaria Haque. Tfx: A tensorflow-based production-scale machine learning platform. In *KDD 2017*, 2017.

[15] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. *CoRR*, abs/1712.05889, 2017.

[16] Taimur Rashid and Nava Levy. Key Takeaways from the First Feature Store Summit, Oct 2021. https://redis.com/blog/feature-store-summit/.

[17] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, 1984.

[18] Anand Lal Shimpi. Intel ssd dc p3700 review: The pcie ssd transition begins with nvme, Jun 2014. https://www.anandtech.com/show/8104/intel-ssd-dc-p3700-review-the-pcie-ssd-transition-begins-with-nvme/3.

[19] Stephanie Wang, Sang Cho, Alex Wu, Clark Zinzow, and Eric Liang. Data processing support in ray, Feb 2021.

[20] Sarah Wooders, Peter Schafhalter, and Joey Gonzalez. Feature stores: The data side of ml pipelines. https://medium.com/riselab/feature-stores-the-data-side-of-ml-pipelines-7083d69bff1c.

[21] Neal E. Young. The k-server dual and loose competitiveness for paging. *CoRR*, cs.DS/0205044, 2002.