Automated Cache Hierarchy for Feature Stores

Project 9: Edward Choi, Shreyas Krishnaswamy, Priyam Mohanty

Problem

- Real-world data pipelines are becoming increasingly reliant on real-time data features
- This has propelled development of feature stores, systems that store featurized data and serve them to ML models or other workloads downstream.
- Many existing feature store are ad-hoc patchworks of existing streaming and storage systems, with some even assuming all workloads fit in memory
- **Goal:** Develop a caching subsystem that leverages unique feature store properties to reduce query latency

Background

- Many modern feature stores are ad-hoc compositions of streaming and storage systems
- Since many existing feature stores are ad-hoc jumbles of systems, they cannot apply end-to-end optimizations across the entire pipeline
- RALF, a feature store from the RISE Lab, leverages a unified pipeline to provide end-to-end optimizations; however, it currently assumes all workloads fit in-memory (it has no system to re-materialize or spill excess features to disk)
- This project uses RALF as a model feature store to explore caching policies that would enable feature stores to support workloads larger than memory while keeping query latency low enough to provide accurate results quickly

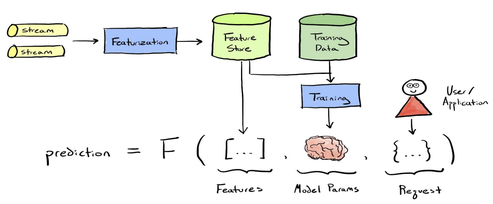

Figure 1: The feature store pipeline takes inputs from various real time sources such as Kafka streams and uses this data to serve inference queries. The feature store also periodically updates training data with new data to keep models up to date.

Wooders, Schafhalter, Gonzalez. (2021, April 6). Feature Stores: The Data Side of ML Pipelines [Illustration]. Feature Stores: The Data Side of ML Pipelines. https://miro.medium.com/max/1400/0*5VqSzEyIoJrA9UwS

Solution

- To benchmark caching policies, the project required adding caching to an existing feature store and comparing query latency. We selected RALF, a unified feature store that assumes all features exist in memory. RALF is built on top of Ray, a distributed systems library. Ray spills objects to disk by LRU as memory is exhausted, so in reality, RALF handles out-of-memory workloads by relying on Ray's object spilling.
- Since much of this project involves exploratory work, rapid prototyping was critical. We built a simulator and defined our own feature store layouts and workloads

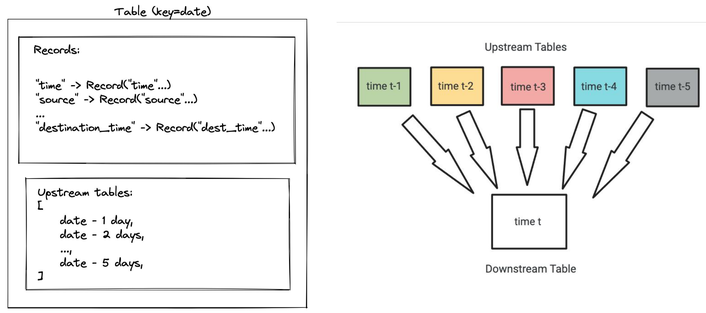

Figure 2: Example workload with daily trips (e.g. Uber) where 5 upstream tables provide data records to a downstream table

- We first equipped the simulator with basic policies like LRU and MRU. Then we tested novel policies based on unique feature store properties.
  - **Cost-awareness**: using latency estimates for rematerialization and upstream feature update frequency, eviction policies can be modified to decide whether to spill to disk or simply evict and recalculate later
  - **Hyperbolic caching**: this is a relatively new caching policy that estimates priority using a mix of time and access frequency. Our simulator extends hyperbolic caching to include cost-aware caching estimates

Results

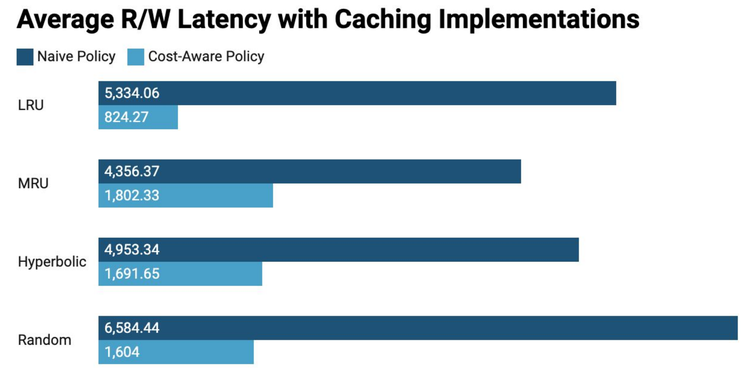

Figure 3: The simulator ran the time-series benchmark from Figure 2 and attained the above latencies. The benchmark contains both reads and writes, and it simulates a commonly used time-series averaging pattern. The values are simulated in arbitrary units of "timesteps" (ts). This workload is the "Non-Linear Multiple Upstreams" workload from Figure 4.

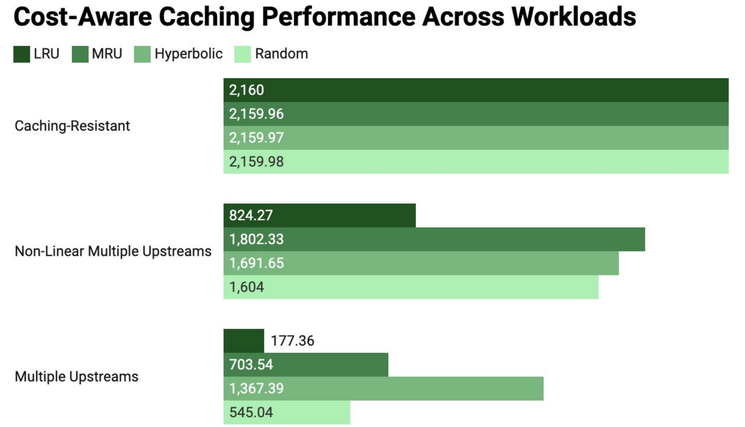

Figure 4: We ran our cost-aware caching methods on three different workloads and measured the "timesteps" for each. *Caching-Resistant*: workload to verify simulator and caching logic. *Multiple Upstreams*: tables with known upstream tables (last 5) accessed in a known order. *Non-Linear Multiple Upstreams*: tables with randomized upstream tables accessed in an unknown order - this is the most realistic "real-world" workload.

Figure 5: The below graph shows the query latency over all requests in the "Non-Linear Multiple Upstream" workload average across either the efficient (i.e. cost-aware) versions of the policy or across the non-efficient (i.e. naive) versions of the policies. The graphs on the right show the non-averaged results to provide more context. The latency is still in "timesteps" whereas the timestamps represent actions in the simulator (either queries or insertions).