Ensuring Data Freshness Across Clouds for Model Serving

Joseph Gonzalez, Ed.
Ion Stoica, Ed.

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2025-36
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-36.html

May 1, 2025

Ensuring Data Freshness Across Clouds for Model Serving

By

Sarah Wooders

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

EECS

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Natacha Crooks, Chair
Professor Ion Stoica
Professor Joseph E. Gonzalez
Professor Vincent Liu

Fall 2024

Ensuring Data Freshness Across Clouds for Model Serving

Abstract

Ensuring Data Freshness Across Clouds for Model Serving

by

Sarah Wooders

Doctor of Philosophy in EECS

University of California, Berkeley

Professor Natacha Crooks, Chair

Modern machine learning applications increasingly rely on large volumes of contextual data, such as pre-computed features and embeddings, to make accurate predictions. However, maintaining the freshness of this derived data across geographically distributed cloud environments presents significant challenges. This thesis explores novel approaches to ensure data freshness for model serving in multi-cloud and multi-region settings, focusing on both feature maintenance and efficient data transfer.

First, we present RALF, a system that optimizes feature updates by leveraging feedback from downstream models to prioritize updates that have the greatest impact on prediction accuracy. RALF introduces the concept of "feature store regret" to quantify the impact of stale features on model performance, enabling more efficient use of computational resources while maintaining high prediction quality.

Building on this foundation, we then address the challenge of efficient data transfer across cloud regions and providers. Skyplane introduces a cloud-aware overlay network that optimizes both cost and throughput for large-scale data transfers. This approach enables faster and more cost-effective replication of feature data across distributed environments.

Finally, we present Cloudcast, which extends the ideas from Skyplane to the multicast setting. Cloudcast leverages cloud pricing models and ephemeral waypoints to minimize the cost of bulk data replication across multiple destinations, further improving the efficiency of maintaining fresh data copies across distributed model serving infrastructure.

Together, these systems form a comprehensive approach to maintaining fresh, derived data for machine learning applications in multi-cloud environments. By addressing both the computational aspects of feature maintenance and the networking challenges of data transfer, this thesis provides a foundation for building more efficient and accurate distributed model serving systems.

To my parents

Contents

List of Figures

List of Tables

Acknowledgments

First, I would like to thank my advisors, Professor Joseph E. Gonzalez and Ion Stoica. I am immensely grateful to have had the opportunity to work with both Joey and Ion throughout my PhD. Joey was always an incredibly supportive and encouraging advisor, who was willing to spend the time to work through low-levels problems, especially early in my PhD. Joey's research group was also exceptionally diverse in its research interests, which allowed me to work with collaborators in both AI and systems research. Ion was also a fantastic advisor to have, with the ability to advise on everything from specific mathematical formulation of problems to strategy around starting companies. I would also like to thank Professors Natacha Crooks, Joe Hellerstein, and Vincent Liu for their mentorship during my PhD, even though they were not my official advisors. I especially am grateful for the support Natacha gave me during difficulties parts of my PhD.

I would also like to my collaborators Simon Mo, Paras Jain, Kevin Lin, Charles Packer, Shu Liu, Shishir Patil, Sam Kumar, and Amit Narang. Simon in paticular was was longest collaborator, and I feel incredibly fortunate to have been able to work with such a talented engineer and researcher throughout my PhD. Kevin Lin was a close second, as we worked together both on the MemGPT and RALF projects together, and I am very grateful for his insight, perspectives, and friendship during the projects we worked. I am also very thankful to Sam Kumar and Paras Jain, who very important mentors to me during my work on Skyplane and Cloudcast. Paras is an incredible researcher and speaker who taught me a lot about how to present my work. Sam provided me with wisdom about how to think of research and the PhD, and also handling paper rejections. I am also grateful to have worked with Shu Liu, who was a very hardworking and talented researcher who helped make the Cloudcast paper happen. Charles Packer has also been an amazing collaborator on the MemGPT project, with incredible sense of optimism, determination, and focus on what's most important.

Outside of collaborators, I am also grateful for students in the EECS program who made the PhD much more enjoyable, such as Audrey Cheng, Jean-Luc Watson, Connor Power, Lisa Dunlap, Suzie Petryk, Justin Wong, Sukrit Kalra, Shadaj Laddad, Samyu Yagati, Alejandro Escontrela, Micah Murray, Asim Biswal, Justin Kerr, Ameesh Shah, Jessy Lin, Ethan Weber, David Chu, Jaewan Hong, Wei-Lin Chiang, Eric Wallace, Ajay Jain, Norman Mu, Frank Luan, Zhanghao Wu, Zongheng Yang, Zhuohan Li, and Justin Kerr. Outside of the PhD, Laura power, Kevin Kwok, Jason Siebel, Claire Nord, Uma Roy, and Rikhav Shah

were also wonderful friends who I am grateful to have had. My roommates, Ameesh Shah, Justin Kerr, and Jessy Lin were an amazing group of friends to spent time with, and also were incredibly supportive during stressful times. Jessy, my friend since freshman year and roommate since graduating undergrad, has been along a parallel path to me for the past five years, exploring startups right out of undergrad, then moving on to pursue a PhD at Berkeley together, and I am grateful to have had her as a friend. I am also thankful to Daniel Rothchild for being an editor to most of what I wrote during my PhD, and also being my co-organizer for the first ever student social retreat (also with Jean-Luc Watson) which led to an annual tradition continued over the past three years.

During my PhD, I also had the opportunity to lead the Computer Science Graduate Entreprenuership club for two years. I am grateful to my co-presidents, Utkarsha Agwan, Shadaj Laddad, and Karl Kauth, and members who helped organize our many events: Jiwon, Shadaj, Charles, J.D., Shishir, Silvery, Marius, and Paras. I am also grateful to Andy Konwinski and Andrew Kriokov for supporting and mentoring the club. I am especially appreciative of the mentorship of Andy Konwinski, who also encouraged me to run the Berkeley LLM meetups in my final year, and was an informal advisor to me throughput my PhD.

I would also like to thank my undergraduate research advisors, Tim Kaler, Professor Charles Lieserson, and Professor Nir Shavit, for encouraging me to attend graduate school and giving me the opportunity to learn to do research.

Finally, I'd like to thank my family: my parents, my brother, and my grandma. Having two professors already in the family (my dad and grandma) made graduate school a much more approachable option, and helped me know what to expect. My mom helped save me a lot of time and stress through helping me move (unfortunately, every year of grad school) and making sure I had nice basics like plates, sheets, and a mattress. My dad is my

Chapter 1

Introduction

Modern machine learning applications increasingly rely on large volumes of contextual data, such as pre-computed features and embeddings, to make accurate predictions. However, maintaining the freshness of this derived data across geographically distributed cloud environments presents significant challenges. This thesis explores novel approaches to ensure data freshness for model serving in multi-cloud and multi-region settings, focusing on both feature maintenance and efficient data transfer.

First, we present RALF, a system that optimizes feature updates by leveraging feedback from downstream models to prioritize updates that have the greatest impact on prediction accuracy. RALF introduces the concept of "feature store regret" to quantify the impact of stale features on model performance, enabling more efficient use of computational resources while maintaining high prediction quality. Building on this foundation, we then address the challenge of efficient data transfer across cloud regions and providers. Skyplane introduces a cloud-aware overlay network that optimizes both cost and throughput for large-scale data transfers. This approach enables faster and more cost-effective replication of feature data across distributed environments.

Finally, we present Cloudcast, which extends the ideas from Skyplane to the multicast setting. Cloudcast leverages cloud pricing models and ephemeral waypoints to minimize the cost of bulk data replication across multiple destinations, further improving the efficiency of maintaining fresh data copies across distributed model serving infrastructure. Together, these systems form a comprehensive approach to maintaining fresh, derived data for machine learning applications in multi-cloud environments. By addressing both the computational aspects of feature maintenance and the networking challenges of data transfer, this thesis provides a foundation for building more efficient and accurate distributed model serving systems.

1.1 Background

Modern machine learning pipelines are increasingly dependent on the freshness of various data components to maintain high prediction accuracy. Two critical elements that require

regular updates are model weights and pre-computed features or embeddings. Understanding the importance of freshness for these components, as well as the challenges in maintaining it, is crucial for building effective and efficient ML systems.

1.1.1 Freshness of Model Weights

Model weights represent the learned parameters of a machine learning model, encapsulating the patterns and relationships extracted from training data. In dynamic environments where data distributions can shift rapidly, maintaining fresh model weights is essential for several reasons:

a) Adapting to Concept Drift: As the underlying relationships in the data change over time (concept drift), models with stale weights may fail to capture new patterns, leading to degraded performance. b) Incorporating New Information: Fresh model weights allow the system to leverage the most recent data, potentially improving prediction accuracy for current conditions. c) Mitigating Bias: Regularly updated weights help prevent the model from becoming overly biased towards historical patterns that may no longer be relevant. However, maintaining fresh model weights faces several challenges:

Computational Cost: Retraining models, especially large ones, can be computationally expensive and time-consuming. Data Availability: Ensuring a consistent stream of labeled data for retraining can be challenging in some domains. Deployment Latency: Updating model weights in production systems often involves complex deployment processes that can introduce delays.

1.1.2 Freshness of Auxilary Data

Pre-computed features and embeddings serve as crucial inputs to many ML models, providing rich, contextual information that can significantly improve prediction accuracy. The importance of their freshness includes: a) Capturing Recent Behavior: In recommendation systems or fraud detection, recent user actions or transactions are often the most predictive. Stale features may miss critical recent events. b) Reflecting Current Context: For applications like news recommendation, embeddings must reflect the current state of rapidly evolving topics and user interests. c) Maintaining Relevance: In domains with high velocity data, such as financial markets or social media, the relevance of features can decay quickly, making freshness critical for accurate predictions.

1.2 Compute Bottlenecks for Freshness

As machine learning models become increasingly integrated into real-time applications, the demand for fresh, accurate features has grown exponentially. This chapter introduces RALF (Real-time, Accuracy and Lineage-aware Featurization), a novel system designed to address the critical challenge of maintaining up-to-date features in high-velocity data environments. RALF represents a significant advancement in feature store technology, focusing

on optimizing feature updates to maximize downstream model accuracy while minimizing computational costs. In modern machine learning pipelines, feature stores have emerged as a crucial component, serving as a central repository for storing and managing pre-computed features. These features, often derived from rapidly changing base data sources, are essential inputs for downstream models, particularly in low-latency prediction serving scenarios. However, existing feature store systems face a fundamental challenge: how to balance the need for fresh features against the computational cost of frequent updates, especially when dealing with complex features like vector embeddings. Traditional approaches to feature maintenance often apply a one-size-fits-all policy, either updating features as frequently as possible (which can be prohibitively expensive) or allowing them to become arbitrarily stale (which can significantly degrade model accuracy). RALF takes a novel approach by explicitly considering the impact of feature staleness on downstream task performance, enabling more intelligent and efficient update strategies. Key contributions of RALF include:

The introduction of "feature store regret," a metric that quantifies the accuracy degradation caused by stale features, providing a principled way to evaluate feature quality. An accuracy-aware scheduling policy that leverages downstream error feedback to prioritize feature updates that have the greatest impact on model performance. A system implementation that enables real-time adaptation to changing data patterns and query workloads, ensuring efficient use of computational resources. Significant improvements in prediction accuracy (up to 32.7

RALF represents a fundamental shift in how we approach feature store maintenance, moving from a data-centric view to an accuracy-centric one. By closely coupling feature updates with downstream model performance, RALF enables more efficient and effective use of computational resources in maintaining feature freshness.

1.3 Network Bottlenecks for Freshness

In the context of maintaining data freshness for machine learning pipelines, network bottlenecks pose a significant challenge, particularly when dealing with large-scale data transfers across cloud regions and providers. Skyplane and its extension, Cloudcast, address these challenges by leveraging cloud-aware overlay networks to optimize both the cost and performance of data transfers. While Skyplane focuses on point-to-point transfers, Cloudcast extends these concepts to multicast scenarios, further improving the efficiency of data replication for distributed ML systems.

Both Skyplane and Cloudcast are built on several key principles that help alleviate network bottlenecks:

1. Cloud-Aware Overlay Networks: Both systems use application-level routing to identify and utilize indirect paths that may offer higher throughput than the default direct path provided by the Internet. This approach allows for more efficient use of available network resources.

2. Cost-Performance Trade-offs: By considering both the monetary cost of data transfer (including egress fees) and the achievable throughput, these systems can navigate the complex landscape of cloud pricing models to find optimal transfer strategies.

3. Resource Elasticity: Both Skyplane and Cloudcast leverage the elasticity of cloud resources, dynamically allocating VMs to serve as relay points in the overlay network. This allows for greater flexibility in routing decisions and can help circumvent bandwidth limitations on individual instances.

4. Mixed-Integer Linear Programming (MILP) Formulation: Both systems use MILP to compute optimal data transfer plans, considering factors such as network throughput, egress costs, and instance limits.

Cloudcast, presented in the NSDI '24 paper, builds upon Skyplane's foundation and extends it to multicast scenarios, further optimizing bulk data replication across multiple destinations. The optimizations provided by Skyplane and Cloudcast have a significant positive impact on maintaining data freshness in distributed ML pipelines:

- Faster Model Weight Updates: By reducing transfer times, these systems enable more frequent updates to model weights across distributed training or serving infrastructure, allowing models to adapt more quickly to changing data patterns.

- More Timely Feature Replication: For pre-computed features and embeddings, the improved transfer speeds and cost efficiency allow for more frequent replication of feature stores across regions, ensuring that serving infrastructure has access to the most up-to-date feature data.

- Improved Multi-Region, Multi-Cloud Deployments: The ability to efficiently transfer data across cloud providers enables more flexible ML pipeline architectures, allowing organizations to leverage the best resources from multiple clouds while maintaining data freshness.

- Cost-Effective Freshness: By optimizing for both cost and performance, these systems make it economically viable to maintain higher levels of data freshness, even for large-scale deployments.

Skyplane and Cloudcast represent significant advancements in alleviating network bottlenecks for maintaining data freshness in distributed ML pipelines. By leveraging cloud-aware overlay networks and sophisticated optimization techniques, these systems enable faster, more cost-effective data transfers, supporting the increasing demands for fresh model weights and features in modern machine learning applications.

Chapter 2

RALF: Accuracy-Aware Scheduling for Feature Store Maintenance

2.1 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 Lee et al. (2019); Guu et al. (2020); Packer et al. (2023); Lewis et al. (2020).

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 Koren et al. (2009); Naumov et al. (2019) 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. 2.1. 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, featur-

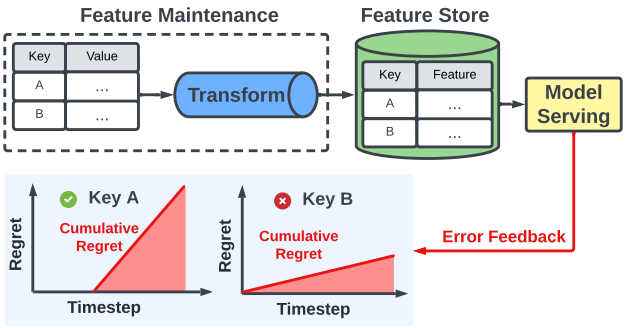


Figure 2.1: Feature stores serve materialized feature values to downstream models. RALF leverage downstream model feedback to prioritize expensive feature updates (§2.3.3).

ization via third party services also may impost hard rate limits on model-based embedding computations OpenAI (2023); Cohere (2023).

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. 2.2. 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 Sivasubramanian (2012); Cooper et al. (2008).

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 **??**. 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

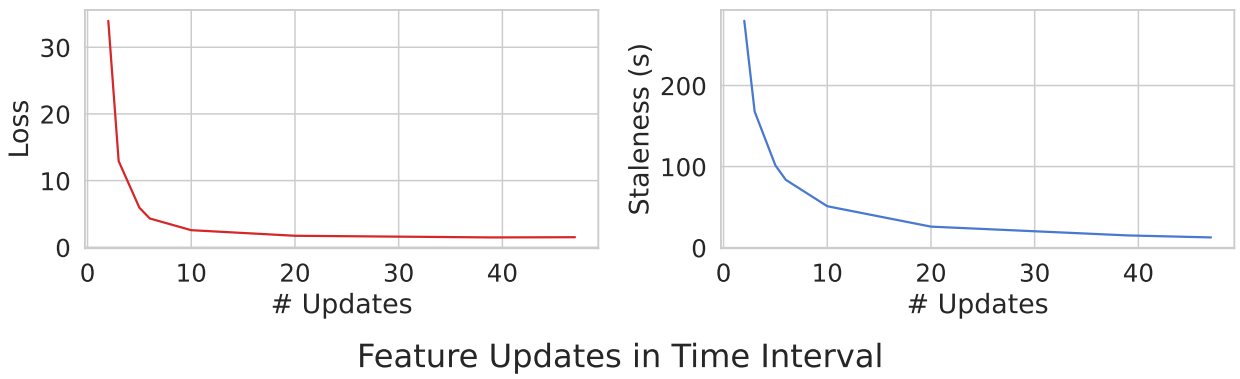


Figure 2.2: The prediction loss (measured by MASE) on the left is correlated with the feature staleness (time since last update), show on the right.

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.

2.2 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,

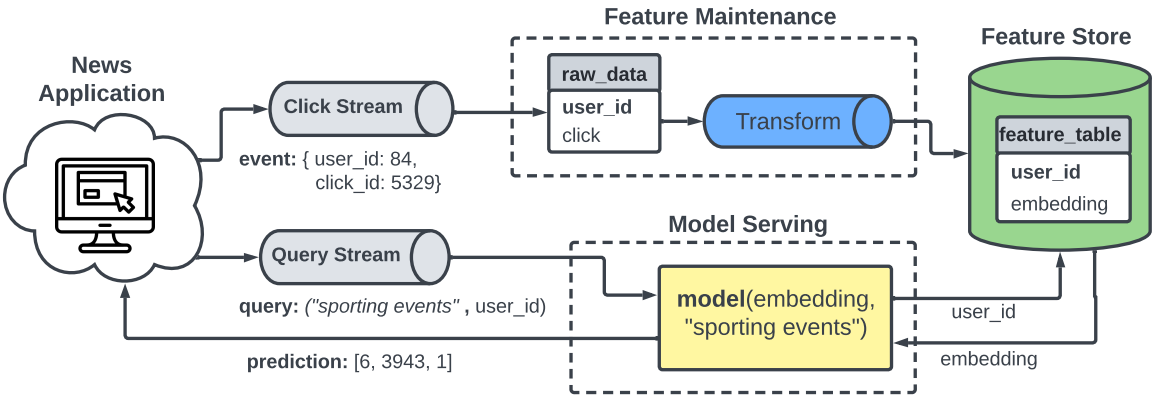


Figure 2.3: A ML serving pipeline with a feature store.

we return to our online news recommendation service example (§2.1).

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

$$\hat{y} = m(x, u, a). \tag{2.1}$$

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:

$$\hat{y} = m\left(x, f_{\text{users}}\left(\mathcal{D}_u^t\right), f_{\text{articles}}\left(\mathcal{D}_a^t\right)\right), \tag{2.2}$$

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 Alteryx (2023). For notational simplicity, in this paper we will focus on a single featurization function f and key k:

$$\hat{y} = m\left(x, f\left(\mathcal{D}_k^t\right)\right). \tag{2.3}$$

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 Naumov et al. (2019).

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*.

2.2.1 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. 2.3. There are several major open-source and commercial feature store systems Hopsworks (2023a); Tecton (2023); Hopsworks (2023b); Services (2023). 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 2.7. 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.

2.2.2 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 Crankshaw et al. (2017, 2020).

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 Mishra et al. (2021), and has developed application-specific solutions for maintaining fresh features for high cardinality data streams Mishra et al. (2021). 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 per-key staleness linearly increasing with time as updates lag new data, as shown in Fig. 2.4. These increases in feature staleness are correlated to decreased prediction accuracy, as shown in Fig. 2.2.

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.

2.2.3 A Feature Store Reference Model

For simplicity, we first describe the standard formulation of a feature store. In Section 2.7, 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:

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

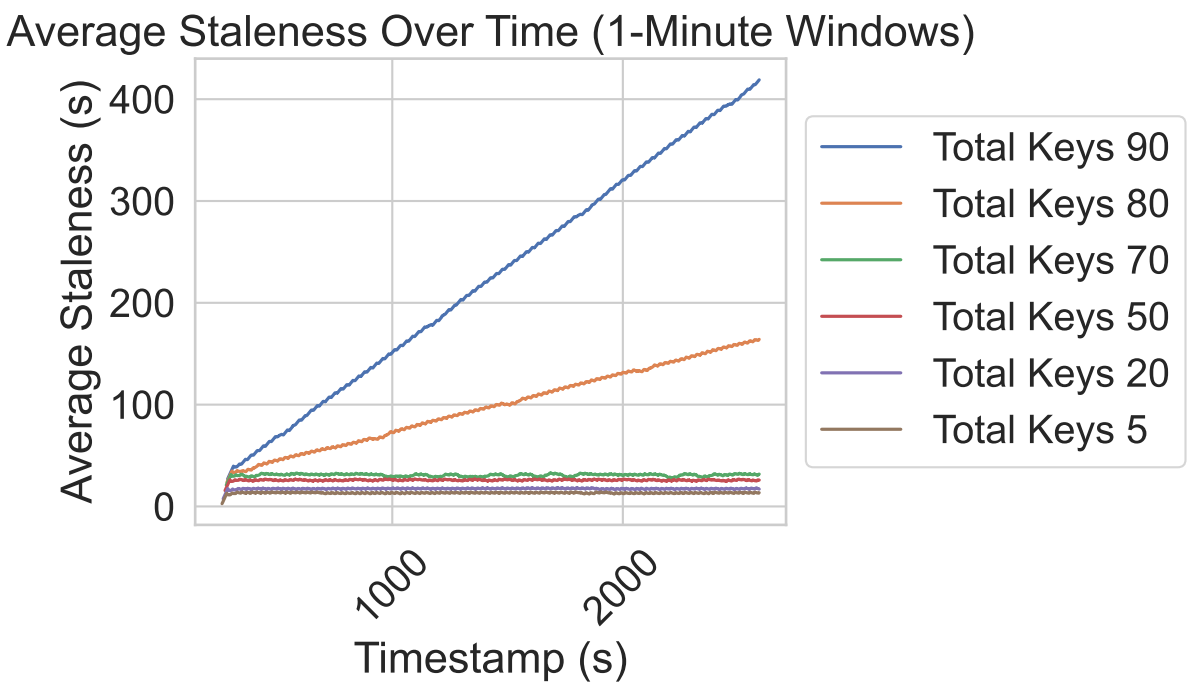


Figure 2.4: Average staleness in a 1-minute time window across all keys as a function as the cardinality.

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:

```
1    SELECT key, uda(data)
2    FROM historical_data
3    WHERE key IN <PolicyQuery>
4    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 2.7, 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 2.7.

2.3 Efficient Feature Maintenance

In this section, we formalize the feature maintenance problem addressed in this paper, that is, selecting the keys for §2.2.3. 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.

2.3.1 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

$$\tilde{v}_k^t = f\left(\mathcal{D}_k^{t-\delta_{k,t}}\right). \tag{2.4}$$

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.

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.

2.3.2 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 $\tilde{v}_k^t$.

$$\mathcal{R}(t) = \mathcal{L}(m|\tilde{v}^t) - \mathcal{L}(m|v^t) \tag{2.5}$$

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:

$$\mathcal{L}(m|v^t) = \sum_i \ell\left(\hat{y}_i(v_t), y_i\right) \tag{2.6}$$

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

2.3.3 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 §2.3.3.

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. (2.4) 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:

$$\arg\min_{\delta} \sum_{t} \mathcal{R}(t) \tag{2.7}$$

$$|U_t| \leq C, \forall t \tag{2.8}$$

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 $\mathcal{L}(m|\tilde{v}_t^k) \approx \sum_{e_k \in E_t} e_k$.

Scheduling Policy

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

$$\arg\max_{k} \sum_{s=0}^{\delta_{t,k}} \mathcal{R}_k(t-s) \tag{2.9}$$

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:

$$\mathcal{R}_k(t) \approx \sum_{e_k \in E_t} e_k - \sum_{e_k \in E_{t_u}} \frac{|E_t| \cdot e_k}{|E_{t_u}|} \tag{2.10}$$

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. (2.9) and denoting the last update time as $t_u = t - \delta_{t,k}$, we select the key to update as:

$$\arg\max_{k} \sum_{s=0}^{\delta_{t,k}} \sum_{e_k \in E_{t-s}} \left(e_k - \sum_{e_k \in E_{t_u}} \frac{e_k}{|E_{t_u}|} \right) \tag{2.11}$$

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.

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.

2.4 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.

2.4.1 RALF Server

RALF orchestrates data updates to maintain feature values. We show an example of defining a maintained feature table with RALF in Listing 2.1. 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 key back to the server. This data is used by the scheduler to help decide which key to update next.

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 2.1. These transformations are implemented as Ray actors, so RALF relies on Ray for concurrency and fault tolerance.

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.

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.

2.4.2 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 2.2, 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.

2.4.3 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 **??**). 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.

2.4.4 Implementation

We construct a full prototype of RALF in about 1,500 lines of Python code. Our prototype is built atop Ray Moritz et al. (2018), because many popular featurization and machine learning libraries (e.g., Tensorflow Abadi et al. (2016)) 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.

2.5 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

Table 2.1: 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 **??**.

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

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.

2.5.1 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 §2.5, which also shows the best and worst-case prediction loss depending on feature quality.

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 Wang et al. (2021); Mishra et al. (2021).

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 Laptev & Amizadeh (2015) and Azure VM dataset Cortez et al. (2017).We use the Python statsmodels library Seabold & Perktold (2010) 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.

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 Grouplens (2023), 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.

2.5.2 End-to-End Evaluation

We evaluate RALF on 800 cores for end-to-end with the Anomaly Detection workload using the Azure VM dataset Cortez et al. (2017). We run RALF with both our Regret-Proportional policy and baseline policy of Round-Robin scheduling to evaluate prediction accuracy, scheduling overhead, and scaleability.

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 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.

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 **??**. 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 **??**. 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 §2.3.2) 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 **??**) enabled by unlimited compute resources (i.e. processing every possible update). We see that the MASE for optimal features and the Regret-Proportional policies are similar in **??**. 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.

Scaling Evaluation

We evaluate how RALF's throughput scales in **??** 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.

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 **??**. 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.

2.5.3 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.

Policies

We implement the **Regret-Proportional** policy described in §2.3.3. 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).

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 **??**.

Regret-Proportional Policy

The Regret-Proportional policy is able to achieve better error across different workloads and numbers of udpates, as shown in **??**. Query-Proportional updates improves error over baseline policies for the Anomaly Detection workload, as shown in Figure **??**. 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. 2.8. 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.

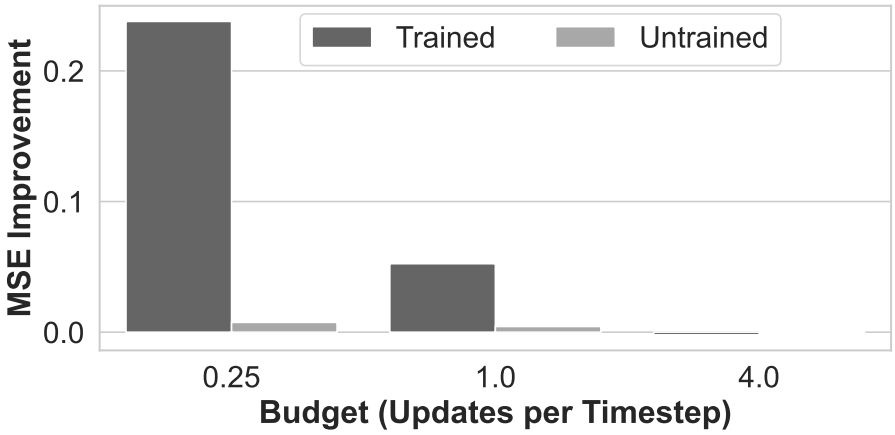


Figure 2.8: Regret-Proportional Improvement over Minimum-Past for users in the training set (Trained) versus new users (Untrained) in the Recommendation workload.

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 **??**. The Regret-Proportional policy is able allocate more updates to features incurring regret the most rapidly, resulting in large update variations.

Optimizing Feature Staleness versus Feature Quality

Although the staleness of the features is correlated to the prediction accuracy as shown in Fig. 2.2, 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 **??**, 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 **??**. Although staleness is strongly correlated to feature quality, optimizing for staleness does not always have the same results as directly optimizing for feature quality.

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 **??** that this leads to similar results as the original distribution of data, showing that Regret-Proportional scheduling is robust to different query

distributions.

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

2.5.5 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 §2.3.3, 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.

2.6 Related Work

Feature Stores. While industry has heavily adopted the use of feature stores Tecton (2023); Hopsworks (2023b), academic research on these systems is limited, and remains focused on metadata and lineage management Kakantousis et al. (2019). We discuss feature stores in depth in §2.7.

Approximate Query Processing. Approximate query processing reduces the cost and latency of queries by returning approximate results Agarwal et al. (2013); Chaudhuri et al. (2017). In the machine learning context, recent work investigates how cheaper and more

expensive models can be combined to respond to queries Kang et al. (2017) while providing formal bounds on approximation Kang et al. (2020). 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 Chirkova et al. (2011) over raw data sources. Prior work in incremental view maintenance and partial view maintenance Zhou et al. (2005) have examined how to efficiently maintain views over changing data. Noria Gjengset et al. (2018) uses partial state and eventual consistency to efficiently materialize tables both on events and on queries. Timely Dataflow Murray et al. (2013) leverages shared arrangements McSherry et al. (2020) 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 Crankshaw et al. (2017, 2020) 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 Kraft et al. (2019), or target specific application use cases such as video analytics Li et al. (2020); Jiang et al. (2018); Bhardwaj et al. (2020).

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 Cooper et al. (2008); Lloyd et al. (2011); Sivasubramanian (2012); Lloyd et al. (2011); Bailis et al. (2012); Terry et al. (2013); Yu & Vahdat (2000); Cui et al. (2014). The flip-side is the increased programmer burden in defending against the potential unexpected application behaviours that arise from these relaxed guarantees Crooks et al. (2016); Bailis et al. (2012). To minimize this issue, prior work either 1) distinguishes between operations whose ordering can be safely relaxed Li et al. (2012); Kraska et al. (2013); Bailis et al. (2014) 2) bounds divergence from the true value when possible Yu & Vahdat (2000); **?**); Wu et al. (1992); Cui et al. (2014); Wong & Agrawal (1992). 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.

2.7 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.

2.7.1 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 Kraft et al. (2019) 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.

2.7.2 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 Redis (2023)).

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.

2.7.3 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.

```
# 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 2.1: Defining a maintained feature table of user embeddings with RALF.

```
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
        )
```

Listing 2.2: Example of a downstream application serving predictions using queried feature values and posting error feedback once the result is observed.

Chapter 3

Cloudcast: High-Throughput, Cost-Aware Overlay Multicast in the Cloud

3.1 Introduction

Increasingly, data in the cloud must be replicated to multiple cloud providers and different regions within each provider. For example, geo-distributed applications like model serving require model weights or features computed in a single region to be replicated to multiple geographic regions to reduce serving latency for users accross the globe Flinn et al. (2022); Sima et al. (2022); Wu et al. (2013). Data sharing between collaborating organizations using different providers similarly requires replicating data to multiple locations. Finally, the growth of multi-cloud applications that leverage resources from multiple providers is dependent on application data being available across provider boundaries Chasins et al. (2022); Yang et al. (2023); Wu et al. (2013).

Of course, data replication and multicast are not new. Both topics have been extensively studied to optimize throughput and scalability in the context of IP networks, peer-to-peer overlays Flinn et al. (2022); Castro et al. (2003); Kostić et al. (2003); Chu et al. (2001); TorrentFreak (2023), and inter-DC networks Zhang et al. (2018); Fatemipour et al. (2022); Luo et al. (2019); Tseng et al. (2021). However, replication between cloud regions and providers introduces first-order concerns beyond just throughput and scalability. In particular, the *monetary cost* of the transfer is a critical factor and one that (as we show later in this paper) is poorly handled by existing techniques for optimizing throughput Zhang et al. (2018); Kostić et al. (2003); Luo et al. (2019). While some existing works consider the monetary cost for multicast, they either ignore the throughout García-Dorado & Rao (2015) or assume a capacity-based pricing model Luo et al. (2021) which is inconsistent with today's cloud. In contrast to capacity-based pricing, cloud providers charge per-GB network egress fees for data transferred out of a given region to another region or cloud provider. Per-GB egress fees introduce a multiplicative term into the transfer cost—(egress price)×(amount

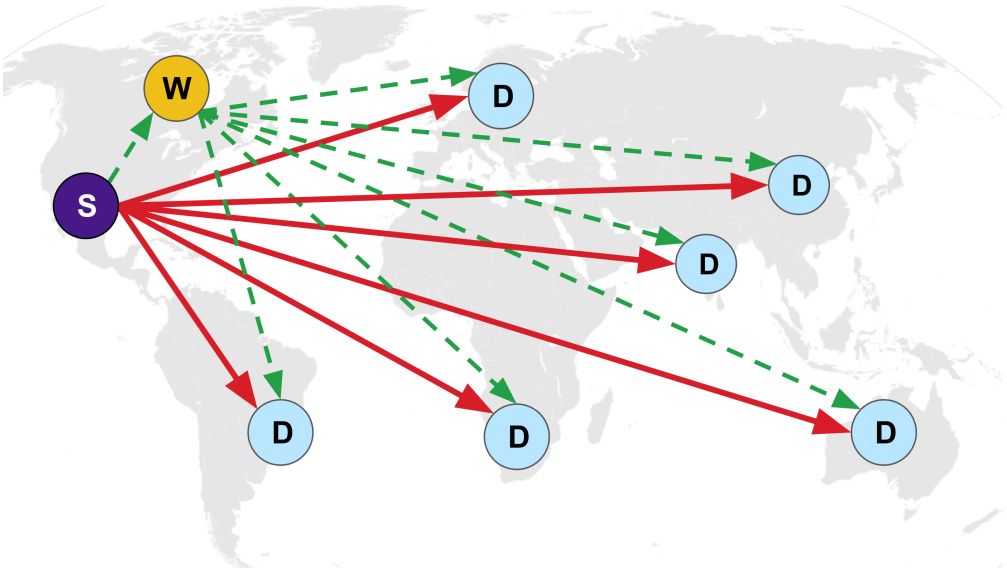


Figure 3.1: Direct replication from a source region (purple) to destination regions (blue) may traverse expensive or slow links, which can be avoided via *waypoint* regions (yellow).

transferred)—making it significantly more difficult to optimize throughput and cost.

Egress costs can vary by orders of magnitude depending on the source and destination Prince & Rao (2021), as well as the capacity of cross-region links. As a result, the structure of the multicast replication tree (i.e., what data is replicated along which paths) can dramatically affect the end-to-end throughput and monetary cost of replication. As a concrete example, consider replication from a GCP source region to six AWS regions (Figure 3.1). Direct replication of the data between the source and each destination region (shown in red arrows) would cost $720 per TB. Instead, replicating to an AWS region with the lowest cross-region egress fees once and multicasting data from that AWS region to other regions (shown in dotted green arrows) would reduce the price to $240 per TB. Further modifying the multicast tree to utilize high-throughput links and offload egress bandwidth from the source node can also improve throughput.

In this work, we solve the problem of *high-throughput, cost-optimized cloud multicast* in which we minimize the cost of data replication while achieving a target replication time (across all destinations) for bulk multicast replication. Cloud multicast incurs costs from network egress fees and compute resources needed to mediate the transfer. In addition, cloud multicast must meet application Service Level Objectives (SLO) for the replication time, such as providing freshness guarantees on replicated data.

We design an optimizer to determine a multicast tree structure given a user-specified source region, destination regions, and target replication time. By providing varying target replication times, our optimizer can generate a Pareto-curve (shown in Figure 3.8) that improves replication cost and throughput compared to prior approaches for cloud multicast García-Dorado & Rao (2015); Ganguly et al. (2005). We achieve this by leveraging techniques such as striping, VM parallelism, and overlay networking, while also accounting for the cloud

providers' network characteristics, resource constraints, and per-GB network pricing model.

Designing this optimization is challenging for two main reasons. First, the optimizer must account for path-specific pricing models, resource constraints, and varying performance across cloud providers. Existing techniques that formulate the optimization problem in terms of bandwidth allocation cannot be adapted to account for per-GB network pricing without making the problem non-linear (described further in §3.3). Second, the optimization search space is combinatorially large, as the optimizer must determine both the set of overlay waypoint regions (regions which are neither the source nor destination) as well as how data will be routed along the overlay network. Unlike the traditional overlay settings, the cloud offers significantly more flexibility in the number and the location of overlay nodes, as cloud VMs can dynamically be instantiated in specified cloud provider regions. Furthermore, replicating subsets of data (i.e., stripes) via different paths is critical for achieving high-throughput Castro et al. (2003). We introduce several approximations (e.g., pre-selecting the regions and limiting path lengths) to reduce this search space and enable the optimizer to run within seconds.

To run overlay multicast across clouds, we develop *Cloudcast*, a system for bulk data overlay multicast across GCP, AWS, and Azure. Cloudcast has a centralized control plane that supports pluggable algorithms for determining the number and location of overlay nodes and replication trees for multiple segments of data. We implement our optimizer as well as several baseline algorithms as part of Cloudcast's control plane. We run system experiments to multicast data across clouds and show that Cloudcast is able to achieve up to 62.4% cost savings and 2.84× replication speedup depending on the control plane algorithm (Figure 3.12).

We run an end-to-end system evaluation comparing Cloudcast with BitTorrent Torrent-Freak (2023) and AWS's commercial offering for multi-region bucket replication Villalba (2020), which, like most cloud data replication offerings, only supports replication into or within that cloud. We find that Cloudcast achieves 7.7× replication speedups and 28.4% cost savings compared to BitTorrent (Figure 3.14). Compared to multi-region bucket replication, we find that Cloudcast achieves up to 61.5% cost reduction and 2.3× replication speedup (Figure 3.13).

To summarize, we make the following contributions:

1. We design an optimizer for minimizing replication cost under replication time constraints.
2. We introduce several approximations to reduce the search space for the optimizer, reducing the solver runtime from hours to seconds.
3. We build Cloudcast, an open-source system for cloud overlay multicast with pluggable data transfer policy.

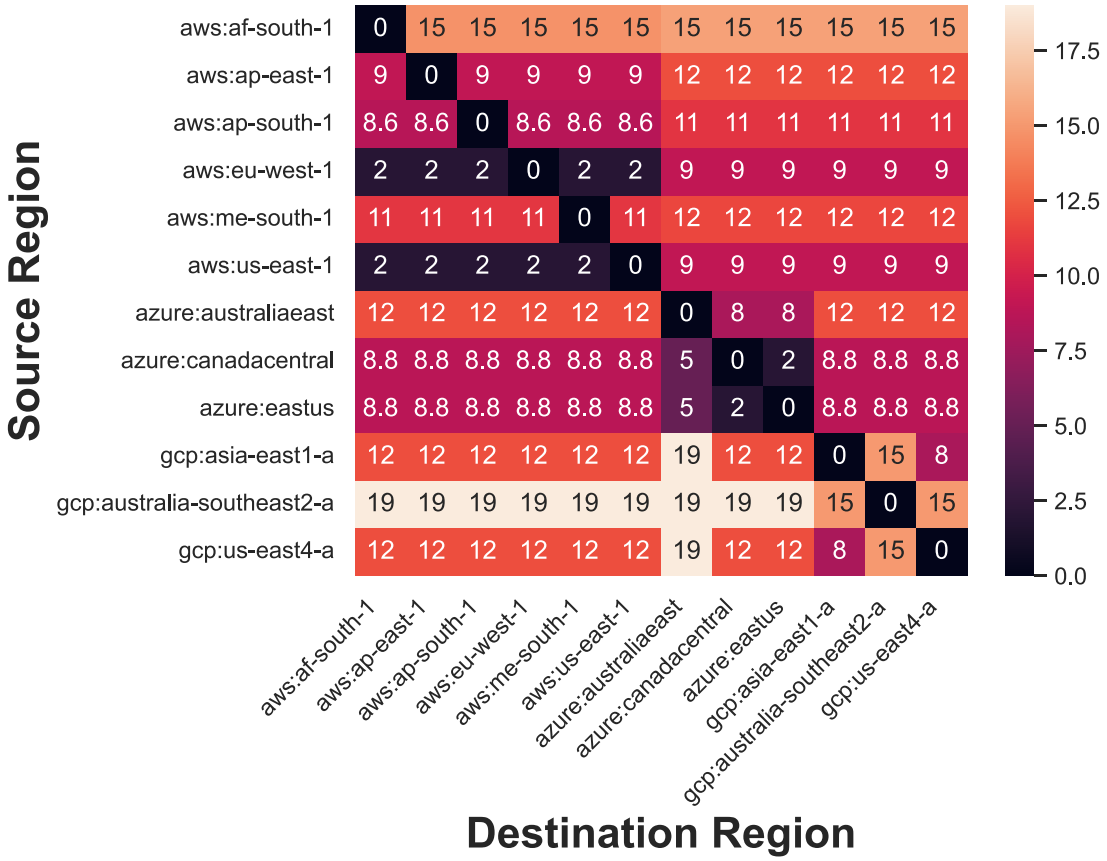


Figure 3.2: Egress fees between regions (in cents per GB).

3.2 Problem Setup

We frame the problem of cloud multicast in terms of constructing an overlay network for replicating data, which involves defining: (1) the set of *overlay nodes* (i.e., cloud VMs) and (2) the *paths* between those nodes that will be included in a multicast replication tree. Cloudcast eventually divides the target data into multiple *stripes* (i.e., partitions), so concurrent replication trees may be used in a single transfer. Our optimization objective is to minimize the monetary cost of replication while also meeting a replication time constraint.

3.2.1 Egress Costs

A unique aspect of multicast in the cloud is the effect of egress costs incurred for data transferred across cloud regions. Cloud providers charge for wide-area data transfer *per-GB* of data transferred. Egress prices—as a method of keeping data within the provider's regions without disincentivizing migration into the provider—dominate data movement costs in the cloud and fundamentally change the multicast problem. Figure 3.2 visualizes the pricing for 11 regions across AWS, Azure, and GCP. Prices vary depending on the source and destination cloud or region, with differences of up to 23× across region pairs. Along those lines, one

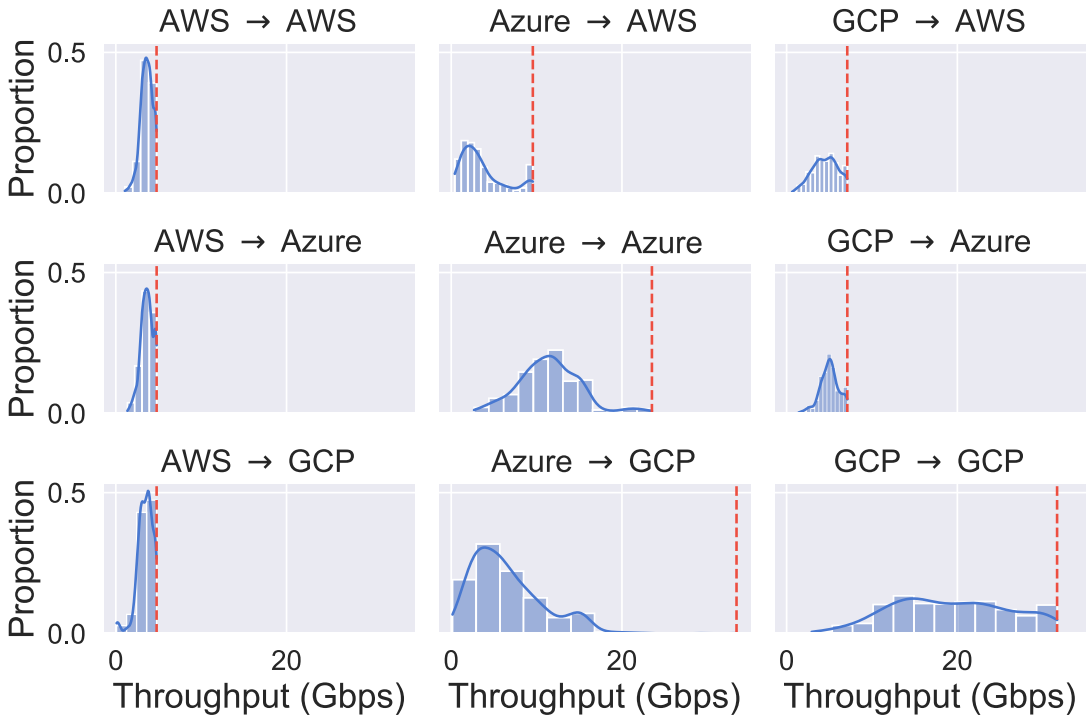


Figure 3.3: Bandwidth distribution (in Gbps) between regions. Per-VM egress limits are marked in red dotted lines.

particularly important axis is whether the transfer stays within a given cloud provider or crosses provider boundaries, as inter-cloud egress costs are generally higher than intra-cloud egress.

Intra-cloud egress (data movement between geographically separated datacenters in the same cloud provider) is priced between $\$0.01 - \$0.19$ per GB transferred. Prices typically increase with longer-distance transfers. For example, GCP charges \$0.08 for transfers between continents but only \$0.02 for transfers within the US. Some smaller providers (e.g., IBM, Cloudflare) offer free cross-region egress.

Inter-cloud egress (data movement between different cloud providers) is typically priced at a much higher rate per GB ($\$0.08 - \$0.23$). As such, it is essential to minimize cross-cloud transfers in a multicast replication tree.

3.2.2 Bandwidth Variability Across Endpoints

Meeting replication time constraints can be challenging due to network bandwidth variability in the cloud. One type of variability arises from cloud providers, who impose constraints on per-VM egress and ingress bandwidth. These constraints differ significantly across providers: for instance, AWS throttles intra-cloud and inter-cloud egress to 5 Gbps per VM, while Azure imposes no VM-level limits. The impact of these egress limits can be observed in Figure 3.3, where bandwidth is capped at the VM egress limit for AWS and GCP. Lim-

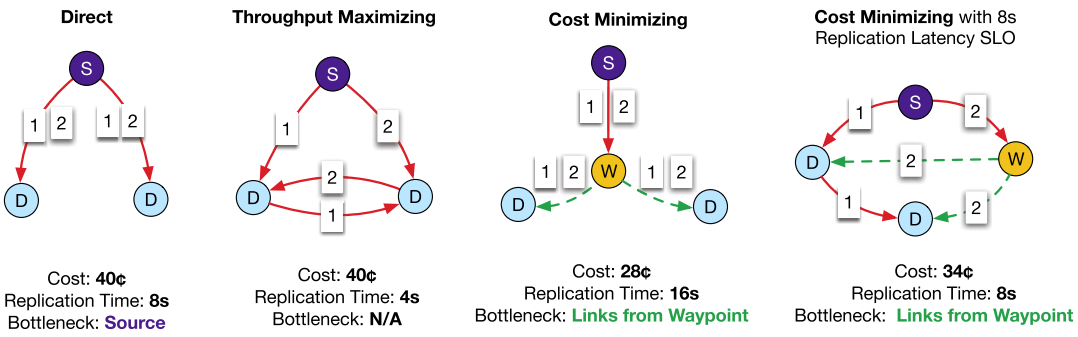


Figure 3.4: Overlay node set and distribution trees for a toy example. The source and destination nodes are marked 'S' and 'D' respectively, while waypoint nodes in yellow are marked 'W'. Expensive, fast paths ($0.1 per GB, 2 Gbps) are shown in solid red, while slow, cheap paths ($0.02 per GB, 1 Gbps) are shown in dashed green.

ited node egress poses a particular challenge for cloud multicast, as the source node's egress bandwidth is often the bottleneck.

Even when source-node bandwidth is not the bottleneck, observed network capacity can also vary considerably across cloud region pairs (up to 202×). Note that these networks are relatively stable across time; prior work Jain et al. (2022) has found that network throughputs are stable over periods of at least 24 hours. Instead, variations are primarily observed across different source and destination regions. Figure 3.3 depicts the distribution of profiled bandwidth between VMs running in AWS, Azure, and GCP. Intra-cloud bandwidth is typically (but not always) higher than inter-cloud bandwidth.

3.2.3 Elasticity of Resources

A major advantage of the cloud is resource elasticity and the ability to flexibly provision VMs across many regions. In the face of the source bottlenecks described above, VM elasticity translates to a corresponding *elasticity of bandwidth*. Allocating multiple parallel VMs enables users to scale throughput beyond per-VM network bandwidth limits.

Unfortunately, adding elastic VM capacity at the source region has limitations. Additional VMs add additional costs due to per-second billing on VMs, which can impact the cost/throughput tradeoff. We note that because the marginal cost of additional VMs is often relatively small compared to egress fees, the tradeoff is often worth making. However even in these situations, bandwidth elasticity has limits: for instance, if a network-based bottleneck is unavoidable or when cloud providers limit the number of vCPUs per region.

Crucially, elastic VM capacity can also be deployed at *waypoint regions* that are neither the source nor the destination. These waypoint regions can help mitigate source VM bottlenecks by distributing load from multicast fan-out across multiple separate regions. Waypoint regions also mitigate points of congestion by routing data around slow paths.

3.2.4 Illustrated Example

Selecting overlay nodes and replication trees to optimize cost and throughput is challenging. Consider the toy example in Figure 3.4 for a 2 GB replication with two 1 GB stripes. Assuming a 4 Gbps bandwidth limit for all nodes and one VM per region, the source ("S") and destination ("D") nodes have fast but expensive outgoing paths, capable of sending at 2 Gbps but costing 10¢ per GB transferred. Other regions have cheaper but slower outgoing paths, capable of sending at 1 Gbps but costing 2¢ per GB transferred. In a simple direct replication scenario, the replication will be bottlenecked by the source node's egress limit (4 Gbps). With two copies of data to send, the total transfer time will be 8 seconds.

Like many bandwidth-optimized techniques Castro et al. (2003); Ganguly et al. (2005); Kostić et al. (2003), we offload egress bandwidth by sending a single data copy from the source and leveraging multiple replication trees. Replication cost is reduced by replicating to a waypoint, and then multicasting to destinations. This doubles replication time to 16 seconds due to stripes being replicated via the slower path (dotted arrows). An 8-second replication SLO is met by transferring just one stripe via the cheaper waypoint.

This simple example presents a large search space for possible replication trees, and real-world cloud networks present additional parameters such as choosing the number of VMs per region and many possible waypoint regions.

3.3 Cost Optimization in Cloudcast

We design an optimizer to minimize replication cost while meeting a replication time SLO (i.e., a constraint on the maximum replication time to a destination). Our optimizer has two main contributions. The first is a Mixed-Integer Linear Program (MILP) formulation of the cost-aware multicast problem, jointly selecting overlay nodes and replication trees. While others Zhang et al. (2018); Ganguly et al. (2005) have used MILP formulations for multicast overlay design, they formulate the optimization problem in terms of *bandwidth*. Extending these formulations to accommodate per-GB costs would violate linearity as data transfer volume (cost) is proportional to the product of the key decision variables: allocated bandwidth and replication time. As a consequence, we propose a new formulation that reframes the optimization in terms of data volume. Our new formulation assigns discrete subsets of data (i.e. stripes) to replication paths in the network while ensuring that a complete copy of the data arrives at all destinations. Unfortunately, solving this MILP formulation can be intractable for larger numbers of destinations. Our second contribution is an approximation of the MILP formulation that significantly reduces solve time without significantly degrading the solution quality.

3.3.1 Egress Cost Minimization Algorithms

The challenge with our optimization problem stems from having to consider *both* throughput and cost. Without replication time constraints, we observe that the Steiner Tree Hwang

Inputs	
TRANSFER-SIZE $\in \mathbb{R}$	*Transfer size in GB*
TIME $\in \mathbb{R}$	*Replication time constraint*
STRIPES $\in \mathbb{Z}_+$	*Number of data stripes*
Decision Variables	
$P \in \{0,1\}^{\lvert\text{STRIPES}\rvert \times \lvert V\rvert \times \lvert V\rvert}$	*Path indicator variable*
$N \in \mathbb{Z}_+^{\lvert V\rvert}$	*Number of VMs per region*
$F \in \mathbb{R}_+^{\lvert\text{STRIPES}+1\rvert \times \lvert V\rvert \times \lvert V\rvert}$	*Flow feasibility variable*
Constants: Cross-Region Paths (edges)	
BANDWIDTH$^{path} \in \mathbb{R}_+^{\lvert V\rvert \times \lvert V\rvert}$	*Bandwidth profile matrix (Gbps)*
COST$^{path} \in \mathbb{R}_+^{\lvert V\rvert}$	*Network cost ($/Gbit)*
Constants: VM Instances (nodes)	
EGRESS$^{VM} \in \mathbb{R}_+^{\lvert V\rvert}$	*Per region per VM egress limit (Gbps)*
INGRESS$^{VM} \in \mathbb{R}_+^{\lvert V\rvert}$	*Per region per VM ingress limit (Gbps)*
COST$^{VM} \in \mathbb{R}_+^{\lvert V\rvert}$	*Per region per VM cost ($/s)*
LIMIT$^{VM} \in \mathbb{Z}_+^{\lvert V\rvert}$	*Max number of VMs per region*

Table 3.1: Symbol table for Cloudcast's ILP formulation.

& Richards (1992) minimizes egress cost. A Steiner Tree is a set of cost-minimizing edges that form a tree that connects a subset of nodes within a graph. If we do not allow the use of waypoint regions, the cost-minimizing tree is a Minimum Spanning Tree (MST). While solving for the MST can be done in linear time, the Steiner Tree problem is NP-hard, though many approximations exist Rehfeldt & Koch (2021). We cannot use the Steiner Tree to account for replication throughput or instance costs, since it only optimizes total edge cost, but we expect our optimizer's solution to be similar to a Steiner Tree in cases where the replication time constraint is loose.

3.3.2 Profiling Cross-region Bandwidth

The bandwidth of paths between cloud regions (both intra-cloud and inter-cloud) is determined by the number of VMs in each region, each VM's egress and ingress limits, and the profiled bandwidth. As discussed in §3.2.2, cross-region bandwidth per VM can be estimated by profiling the bandwidth between region pairs using `iperf3`. Egress and ingress limits vary across cloud providers but are static and can be determined by cloud providers' documentation AWS (2023); Azure (2023); Cloud (2023). We utilize these profiles as an estimate of expected network bandwidth for the duration of a transfer. Profiling results are

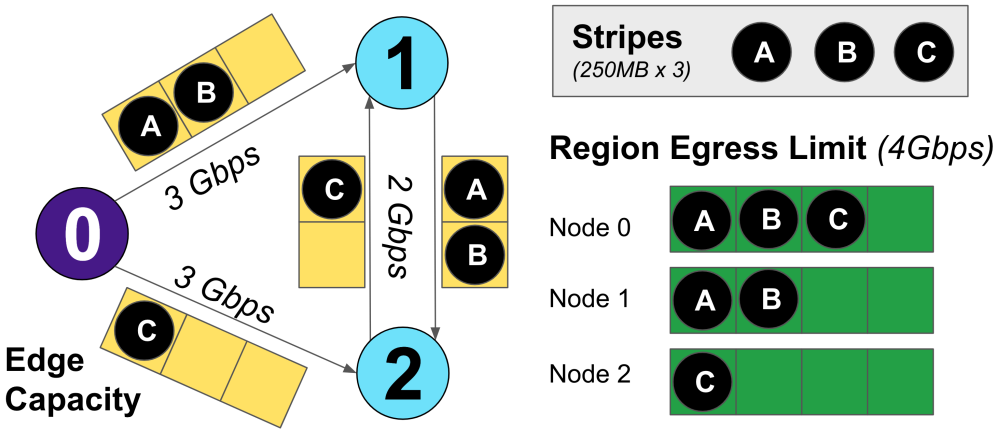


Figure 3.5: Stripes transferred from the source (purple) to destinations (blue) are placed by the solver along edges depending on edge capacity (yellow) and node capacity (green).

included as part of our open-source repository and shared across all users of Cloudcast.

3.3.3 Optimizing Cost with Time Constraints

In order to minimize replication price while meeting runtime requirements and cloud resource constraints, we frame a MILP on a directed graph representing the entire cloud topology. The input to the optimizer is the transfer size: TRANSFER-SIZE, the runtime budget: TIME, and the number of stripes: STRIPES to divide the data into.

To formulate the optimization problem as a MILP, formulate the problem in terms of allocating data *volume* to edges rather than bandwidth, with allocation units per stripe. We translate cross-region bandwidth and per-region egress/ingress limits into volume capacities, as shown in Fig. 3.5, this determines how many stripes can fit along each edge. This makes the MILP similar to a bin packing problem, where we aim to pack stripes into edges such that all destinations receive all stripes. The volume-based representation allows cost to be computed as a function of the number of stripes placed on each edge.

Next, we formally describe the MILP decision variables, objectives, and constraints. The cloud regions and cross-region paths are represented as $G = (V, E)$, where V denotes the set of cloud regions and E denotes paths between regions. We provide a reference table for the notation in Table 3.1.

Decision variables

The MILP formulation consists of three decision variables. The path indicator variable $P_{s,(v,u)}$ indicates whether a stripe s is sent between regions $(u, v) \in V$. The paths selected by P make up the multicast replication tree for each stripe. The decision variable N_v represents the total number of overlay routers in the region v. An additional flow variable $F_{s,(u,v)}$ ensures valid paths when constructing the multicast tree. It ensures that the paths selected by P do not contain cycles and are connected, by allowing flow to be pushed from the source to all destinations for each stripe (see §3.3.3).

Objective: minimizing price under a deadline

To minimize the price of a multicast transfer while meeting replication time constraints, we use a two-part objective function. The first part optimizes the number of virtual machines (VMs) per region, represented by N, and the second part optimizes the distribution trees per stripe, represented by P. The objective is formulated as follows:

$$\arg\min_{P,N} \quad \underbrace{\text{TIME} \times \langle \text{COST}^{VM}, N \rangle}_{\text{Instance Cost}} \tag{3.1}$$

$$+ \quad \underbrace{\frac{\text{TRANSFER-SIZE}}{\text{STRIPES}} \times \sum_{s \in \text{STRIPES}} \langle \text{COST}^{path}, P_s \rangle}_{\text{Egress Cost}} \tag{3.2}$$

The price of a data transfer is the sum of the instance fee and the egress fee. The instance fee depends on the number of VMs running per region, the job completion time, and the per-region VM fee. The egress fees are determined by the data distribution path and the amount of data traversed through the path, as defined by P. We note that the instance cost is also an upper bound as it can be potentially overestimated if the data transfer is completed in less than the user-defined time budget. However, this is necessary to ensure linearity.

Constraints

We represent cross-region bandwidth, node egress/ingress bandwidth, per-region VM limits, and replication tree structure requirements as constraints within the MILP.

Representing Inter-Region & Inter-Cloud Bandwidth. Cross-region bandwidth is represented as the *per-GB capacity* given the run-time budget, i.e., how many stripes can fit along an edge. Increasing the number of VMs in the source regions linearly increases the rate at which we can send data. We thus model the bandwidth between two regions as the per-VM bandwidth profiled between those two regions multiplied by the number of VMs in the source region:

$$\text{CAPACITY}^{path} = \langle N, \text{BANDWIDTH}^{path} \rangle * \text{TIME}, \tag{3.3}$$

and constrain P in terms of the path capacity:

$$\forall (u,v) \in E \qquad \text{SIZE}_{\text{STRIPE}} * \sum_{s} P_{s,(u,v)} \leq \text{CAPACITY}^{path}{}_{(u,v)}. \tag{3.4}$$

to ensure allocated stripes fit within the capacity.

Representing VM Bandwidth Constraints. Cloud providers impose per-VM bandwidth constraints on network egress, as described in §3.2.2. As such, a major bottleneck of multicast transfer is the source region's limited egress bandwidth. We constrain P in terms of the

ingress and egress limits:

$$\forall v \in V \quad \text{SIZE}_{\text{STRIPE}} * \sum_{s} \sum_{u \in V} P_{s,(v,u)} \tag{3.5}$$

$$\leq \text{EGRESS}^{VM}{}_{v} * N_v * \text{TIME} \tag{3.6}$$

$$\forall u \in V \quad \text{SIZE}_{\text{STRIPE}} * \sum_{s} \sum_{v \in V} P_{s,(v,u)} \tag{3.7}$$

$$\leq \text{INGRESS}^{VM}{}_{u} * N_u * \text{TIME} \tag{3.8}$$

Representing VM Capacity Constraints. We account for per region VM limits by adding the constraint $N \leq \text{LIMIT}^{VM}$.

Ensuring Valid Multicast Trees. We use an additional variable F to ensure that the paths selected by P are valid distribution trees, i.e., they are connected and acyclic, and they deliver all data to each destination. At a high level, we ensure that $F_{s,(u,v)} \geq 1$, if $P_{s,(u,v)} = 1$, and impose conservation of flow constraints on F but not P, since P is an indicator variable not a flow variable. We then ensure that flow can be pushed from the source node to destination nodes on F for each stripe, which also ensures that flow can be pushed from the source to destination for the paths selected by P (without having to impose flow conservation on P). We leave details on this part of the formulation for **??** due to space.

Solver feasibility

Our formulation so far has a search space of size $O(2^{|V|^2 \times |\text{STRIPES}|})$. With 71 possible regions across GCP, AWS, and Azure and 10 stripes, the search space is, therefore, $O(2^{50410})$, which is infeasible even for advanced solvers to solve within a few minutes, necessitating approximations.

3.3.4 Reducing Optimizer Runtime

In this section, we describe several mechanisms that we combine to reduce the optimization runtime or an order of seconds, while still maintaining solution quality.

Node Clustering. We observe that many regions across cloud providers share similar characteristics in terms of bandwidth and the costs of their outgoing and incoming paths. A motivating observation was that sub-sampling regions randomly could produce similar solutions with much lower solve time, as shown in Figure **??**. At a high level, AWS regions in Europe regions all have similar egress/ingress costs and bandwidth, so only one of those regions needs to be considered as a potential waypoint. Therefore, to reduce the optimizer search space, we cluster regions using their incoming and outgoing path costs and bandwidth as features and select a representative node from each cluster. We empirically find that, with about 20 clusters (i.e. 20 subsampled regions), the optimizer can generate solutions that are reliably similar to the original MILP without approximation (more discussion in §3.5.3).

Hop Constraining. To further reduce the optimization space, we only consider a maximum of 2-hop overlay waypoints. Previous research has shown that limited numbers of

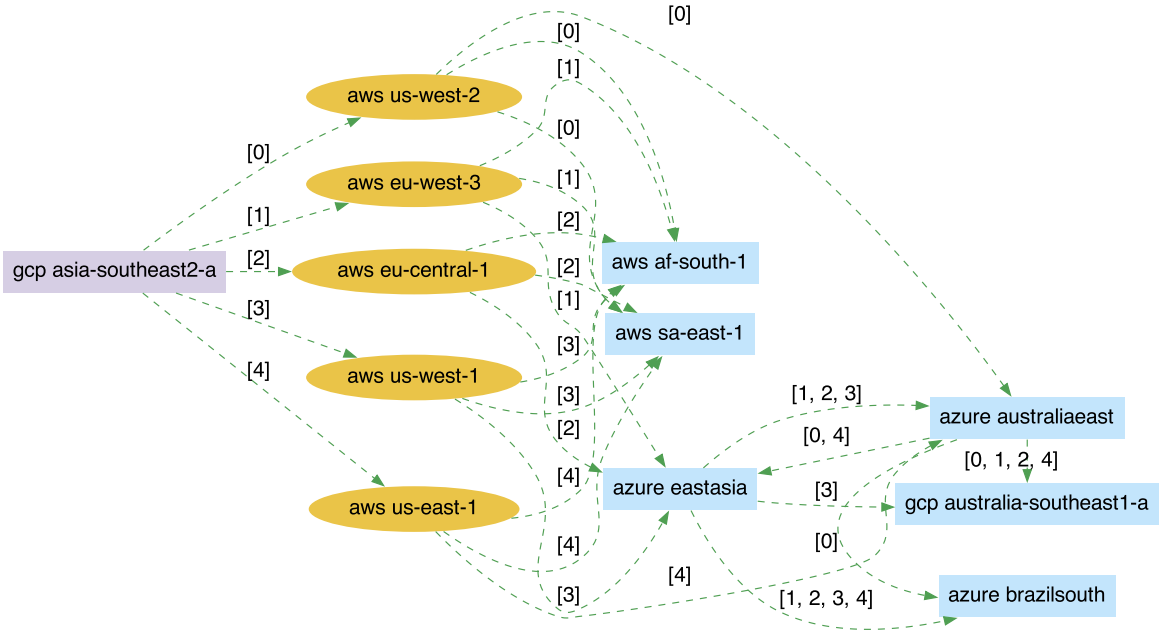


Figure 3.6: Visualized solver output for inter-cloud replication described in §3.5.1, consisting of source (purple), waypoint (yellow), and destination (blue) regions. The data is divided into 5 stripes (marked on edges).

overlay hops are often sufficient Andersen et al. (2001); Peter et al. (2014); Stoica et al. (2002). Our analysis also found solutions using multiple overlay hops to be rare, suggesting that they need not be considered. We implement the hop constraints as an additional constraint on the MILP.

Stripe-iterative Approximation. To make the optimizer runtime linear with respect to the number of stripes (rather than exponential), we design a greedy, stripe-iterative approximation algorithm that solves for one stripe per iteration. We solve for each stripe independently, then update the input graph for the next stripe by reducing the path capacity (CAPACITY^{path}), instance limits, and egress/ingress limits per region (LIMIT^{VM}, EGRESS^{VM}, and INGRESS^{VM}).

3.3.5 Example Topology

We show an example of the optimizer's output replication tree topology visualized in Figure 3.6. Due to variability in cloud provider egress pricing and cross-region throughput, our optimizer often finds unexpected solutions, such as routing one stripe (marked `[3]`) from GCP to AWS, AWS to Azure, then back to GCP. Although questionable at first glance, we evaluate this same replication in Figure 3.12 and demonstrate both cost and replication time improvements over baselines.

3.4 Architecture of Cloudcast

A key contribution of this work is the design and implementation of the Cloudcast artifact, which provides a practical, performant, and extensible system for studying overlay multicast algorithms in cloud environments. The Cloudcast system simplifies the design and deployment of multicast overlays spanning cloud object stores. We use it to implement and deploy the optimizer described in §3.3 and several baseline algorithms.

We provide an overview of Cloudcast in Fig. 3.7. Cloudcast is designed with a centralized control plane and a distributed data plane. The control plane determines the set of overlay nodes and routing paths, and it dispatches and monitors multicast jobs. The data plane consists of *overlay routers*, which we implement as modular software routers running on overlay nodes deployed on cloud VMs. The control plane configures overlay routers using a *router program*, which specifies a graph of modular operators for processing data. Each overlay router gets a unique router program, which, in cooperation with other routers in the system, implements the desired flow of data over the overlay network.

Cloudcast is implemented as part of the Skyplane Jain et al. (2022) open source project and consisted of additional lines of Python to implement the .

3.4.1 Control Plane

The control plane contains the planner, which supports pluggable algorithms for determining the placement of overlay routers across cloud providers and paths along which data is

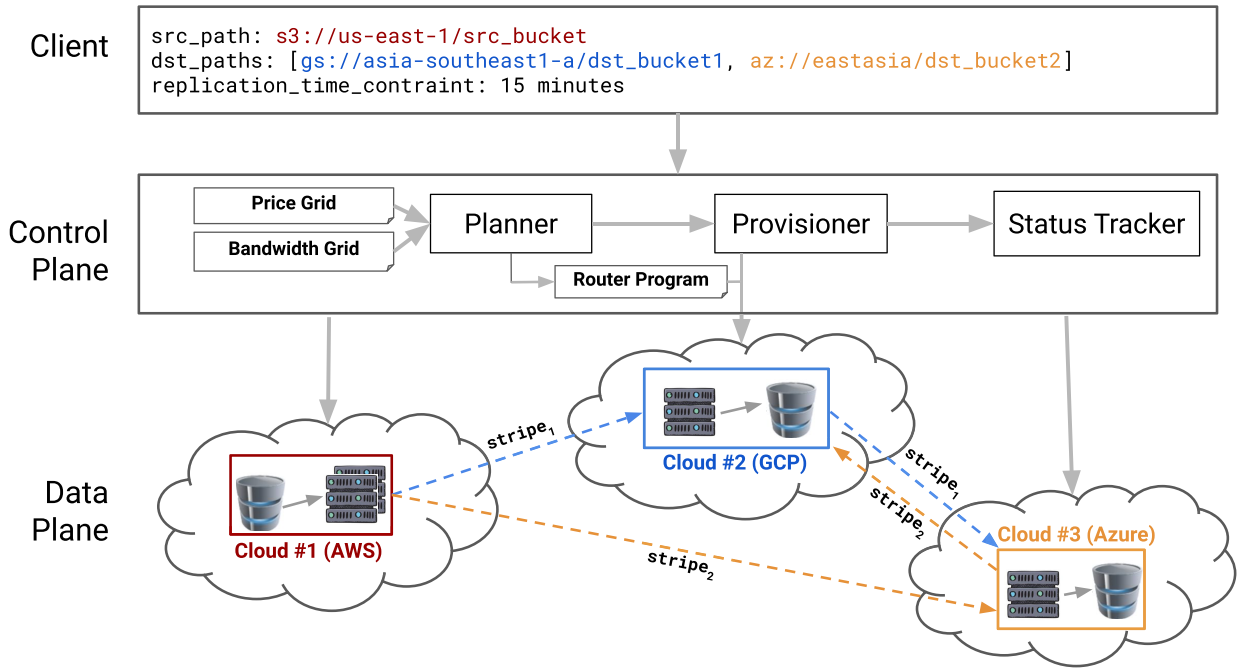


Figure 3.7: Cloudcast system architecture.

replicated (shown in Fig. 3.7). The output of the planner is used to provision VMs to act as overlay routers across cloud regions and to compile a router program for each overlay router that configures its behavior. Finally, the control plane initiates the transfer and monitors its progress.

Planner. The planner is responsible for creating a multicast plan based on a target replication time, source and destination object store paths provided by the user, and profiling data described in §3.3.2. The planner takes as input the algorithm to use for generating a multicast plan, which can be the default Cloudcast optimizer described in §3.3.3 or a custom plan (e.g., a Steiner Tree over the cost graph). The planning algorithm determines how many overlay nodes to create in each region and how each data stripe should be routed through the overlay network. The planner uses the algorithm output to generate a *router program* for each overlay router, which specifies how the overlay router should process a chunk header when received. The Cloudcast default optimizer is implemented using Python's CVXPY library cvx (2021) (version 1.3.2) with a Gurobi solver Gurobi Optimization, LLC (2023), implemented in about 1K lines of code.

Provisioner. Once a multicast plan is determined, the provisioner instantiates the overlay routers. The provisioner creates a VPC in each cloud provider and provisions VMs to act as overlay routers within these VPCs. The provisioner also sets firewall rules to allow network traffic between overlay routers, which send and receive data from each other, as specified by the planner-generated router programs. Once a VM has been instantiated, the provisioner installs and launches the router programs as containers on the VMs.

Chunk Dispatching and Status Tracker. The control plane subdivides replication target data into *chunks*, which are at most 64MB in size, to allow for transfer pipelining and parallelism. Each chunk has a *chunk header*, which specifies a key (e.g., object store object, filename), byte range, and an optional multipart ID (required for multipart uploads). The chunk header also contains a *stripe ID*, which specifies which path along the overlay the chunk will take.

The control plane informs each source overlay router (i.e., overlay routers responsible for reading source data) the chunks for which they are responsible by sending the corresponding chunk headers. We refer to this as *registering* a chunk to an overlay router. The control plane's status tracker monitors the status of each chunk by querying the status of chunks on each overlay router.

3.4.2 Data Plane

The data plane is composed of overlay routers, each running on a single VM. The overlay routers are created and configured by the control plane to execute the transfer according to the multicast plan. Cloudcast supports configurable overlays by defining processing on overlay routers using modular operators, inspired by the design of configurable routers Kohler et al. (2000).

The router program provided by the control plane specifies a directed acyclic graph (DAG) of *operators* (analogous to elements) and *connections*, all of which run on each overlay router and are used to process incoming chunk headers registered to the overlay router. The DAGs are created at the overlay router's startup time based on the router program, and they allow overlay routers to process chunks without additional coordination with the control plane.

Operators are implemented as a pool of worker processes running processing steps for a chunk, such as reading the chunk from the source object store, relaying the chunk to another overlay router, writing the chunk to a destination object store, or transforming the chunk data (e.g., compression or encryption). Connections pass chunk headers between operators via thread-safe queues, and can be configured to send a chunk header to one or all of multiple downstream operators.

For example, on a source overlay router, chunk registrations from the control plane will provide chunk headers to the first operator in the DAG, which downloads chunk data from an object store. All chunk data is stored in a shared memory filesystem to allow for fast access across operators. Once chunk data is downloaded, the chunk header is passed to the next operator via a connection, which runs LZ4 compression lz4 (2023)) and secret key encryption pyn (2023); Denis (2013) on the chunk data. The leaf operators are 'sender' operators, which relay the chunk header and data to other overlay routers.

Chunk data is relayed between overlay routers by a 'sender' operator on the sending router and a 'receiver' operator on the receiving overlay router. When the sender operator is created, it creates parallel TCP connections which are kept open for the duration of the transfer. Before sending chunk data, the sender will attempt to register the corresponding

System	Description
Direct	Data is transferred directly from the source to the destination regions.
MDST	Data is transferred along edges selected by a Minimum Directed Spanning Tree (including source and destination regions) computed from network costs.
Steiner Tree	Data is transferred along edges selected by a Steiner tree (including optional waypoint regions) computed from network costs.
SPIDER	Data is transferred according to the plan generated by SPIDER, a system designed for fast bulk replication to multiple destinations.
Skyplane	Skyplane's optimizer is used to select paths for each source-destination pair, which are combined to build the distribution tree.
CloudMPCast	Data is transferred over a set of cost-minimizing edges that meet a minimum bandwidth threshold.
Deadline-aware Inter-DC Multicast	Data is transferred to meet deadlines in the inter-DC context according. Note that due to scalability issues, we needed to modify the candidate tree generation step to only consider a subset of waypoint regions to achieve tractable runtimes.
AWS S3 Multi-Region Bucket	Vendor product that supports intra-cloud between AWS regions only. We enable Replication Time Control.
Bullet	Data is transferred according to the plan generated by Bullet, a high-bandwidth dissemination technique using an overlay mesh.
BitTorrent	Peer-to-peer protocol where peers download data from each other in a decentralized manner.
Cloudcast-Opt (HT)	Data is transferred along the highest throughput (HT) multicast tree generated by our optimizer (tightest time constraint).
Cloudcast-Opt (LC)	Data is transferred along a low cost (LC) multicast tree generated by our optimizer (relatively loose time constraint).

Table 3.2: All of the systems and variants we evaluate, covering a mix of academic baselines and commercial solutions.

chunk headers with the receiving overlay router to ensure it has space in its shared memory file system to write the chunk data. Once chunks are registered, the sender will send the chunk data over the TCP sockets, and the receiver will wait for the written chunk data size to match the size specified by the chunk header, before sending chunk headers to the next operator. Successfully sent chunk data is deleted from the shared memory filesystem.

Backpressure. Connections are configured with a maximum size for the underlying queues. If the queue reaches its maximum size, the upstream operator will wait until the queue size decreases sending chunk headers to the connection.

Striping. Registered chunk headers with different stripe IDs are placed in different queues and processed by separate DAGs, so that different stripes can be routed differently.

3.5 Evaluation

In this section, we evaluate Cloudcast across three metrics: replication cost, replication time, and the optimizer solve time (or simply, runtime). In particular, we show that for

intra-cloud and inter-cloud bulk data transfer, Cloudcast is able to achieve up to 61.5% cost improvements under a tight runtime budget when compared to academic, commercial, and open-source baselines. We also show that our approximations to reduce the optimizer solve time (as discussed in §3.3.4) are highly effective by reducing the runtime by, on average, 30.68× for 5-destination replications. To simplify evaluation, we disable compression and encryption in experiments.

The full list of evaluated baselines is shown in Table 3.2. We note that many algorithms do not determine the number of VMs to use in each region. To present them in the best light possible, we maximize the number of VMs in each region traversed by data, subject to per-region quota limits.

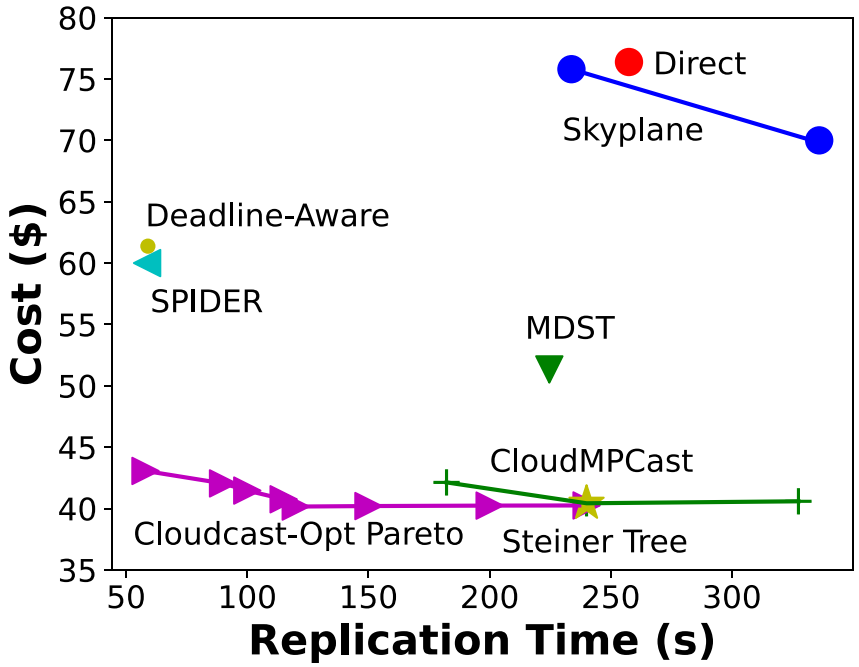


Figure 3.8: Simulated results for Multicast Algorithms.

3.5.1 Comparison to Multicast Algorithms

We compare the replication time and cost of existing multicast algorithms with Cloudcast's optimizer to send 100 GB of data from one source to six destination regions.

Simulation results. Given the above replication scenario, we start by exploring a wide range of algorithmic baselines and Cloudcast parameter settings through simulation. While we tested many configurations through the development of Cloudcast, due to limited space, we present results for a representative configuration[1]. Evaluated systems include Cloudcast-Opt, direct transmission to the destinations, sending along cost-minimizing trees (MDST and

[1]Simulated Inter-Cloud: from `gcp:asia-southeast1-a` to `azure:eastasia`, `aws:af-south-1`, `azure:brazilsouth`, `aws:sa-east-1`

Steiner Tree), SPIDER Ganguly et al. (2005), CloudMPCast García-Dorado & Rao (2015), Skyplane Jain et al. (2022), and a deadline-aware inter-DC optimizer Ji et al. (2018). Although Skyplane's optimizer is designed for unicast, not multicast, we adapt the optimizer's solution to multicast by running the optimization for each source-destination pair, and then combining all the graphs to build the distribution tree.

For Skyplane, CloudMPCast, and Cloudcast-Opt, we vary the throughput parameter to evaluate the performance range. For CloudMPCast García-Dorado & Rao (2015), the optimizer allows for the level of throughput degradation to be controlled by an $\alpha \leq 1$ term, which determines how aggressively edges are filtered out. Our parameter sweep includes $\alpha \in [1, 0.5, 0.1]$, where $\alpha = 1$ maximizes CloudMPCast's throughput. For Skyplane, we vary the target throughput to maximize throughput and minimize cost, and plot both of these points. For Cloudcast-Opt, we show results for several replication time constraints.

In Fig. 3.8, we see that all baselines improve significantly upon direct transmission, and while some can match Cloudcast-Opt's capacity for fast replication time or low cost, no existing baseline can optimize both metrics simultaneously. Rather, Cloudcast-Opt's Pareto-curve can match or beat all baselines on at least one of cost or performance. CloudMPCast, whose α parameter does provide some flexibility, still offers a worse tradeoff than Cloudcast-Opt. Skyplane also has a significantly worse tradeoff curve, as it is not designed for multicast, so does not perform optimizations to alleviate source bottlenecks which are crucial for achieving high throughput. Despite this, even Skyplane's can improve throughput (for the throughput-maximizing solution) and reduce cost (for the cost-minimizing solution) as compared to direct transfers.

Cloud deployments. The remainder of our evaluations present empirical results from real cloud data transfers. Due to the high cost of running data multicast in the cloud ($20–$110 per transfer), we limit our evaluation to four representative configurations and four representative baselines identified by our simulation results. Among the configurations, three are intra-cloud replications corresponding to AWS[2], Azure[3] and GCP[4], and one is an inter-cloud replication workload that covers all three major providers[5]. These configurations are chosen to contain a source region with high egress costs to demonstrate potential cost savings. Among the baselines, we sub-selected the best-performing baselines from our simulation results in terms of throughput (SPIDER) and cost (Steiner Tree), with direct transmission providing a naive baseline.

Fig. 3.9, Fig. 3.10, Fig. 3.11 show results for AWS, GCP, and Azure intra-cloud replication, and Fig. 3.12 shows inter-cloud results. Across all configurations, given a very tight

[2]AWS Intra-Cloud: from `ap-east-1` to `us-west-1`, `ap-northeast-3`, `eu-north-1`, `ap-south-1`, `ca-central-1`, `ap-northeast-1`

[3]Azure Intra-Cloud: from `brazilsouth` to `westeurope`, `westus`, `koreacentral`, `australiaeast`, `uaenorth`, `centralindia`

[4]GCP Intra-Cloud: from `asia-southeast2-a` to `australia-southeast1-a`, `southamerica-east1-a`, `europe-west4-a`, `europe-west6-a`, `asia-east1-a`, `europe-west2-a`

[5]Inter-Cloud: from `gcp:asia-southeast1-a` to `azure:australiaeast`, `azure:eastasia`, `aws:ap-southeast-2`, `azure:brazilsouth`, `aws:sa-east-1`, `gcp:australia-southeast1-a`

replication time constraint, Cloudcast-Opt (HT) solution leads to $46-62.4\%$ cost reductions and $2-2.84\times$ replication time speedup compared to sending directly to each destination.

Of the baselines tested, SPIDER Ganguly et al. (2005) consistently demonstrates the lowest replication time, as it did in simulation. However, as SPIDER is not cost-aware, Cloudcast-Opt (HT) can achieve $28.4-44.0\%$ cost savings. Surprisingly, while saving significant cost, Cloudcast-Opt (HT) simultaneously speeds up replication by $1.11-1.35\times$, beating SPIDER on both axes. If, on the other hand, Cloudcast is given a loose replication time budget, i.e., Cloudcast-Opt (LC), it can find the cost-optimal solution in all setups, matching Steiner Tree solutions.

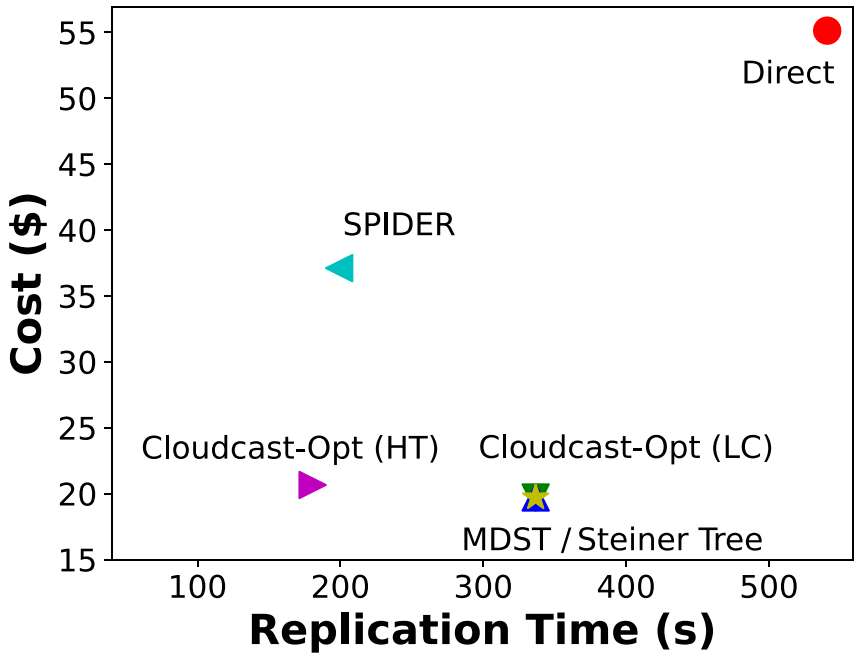


Figure 3.9: AWS Intra-Cloud

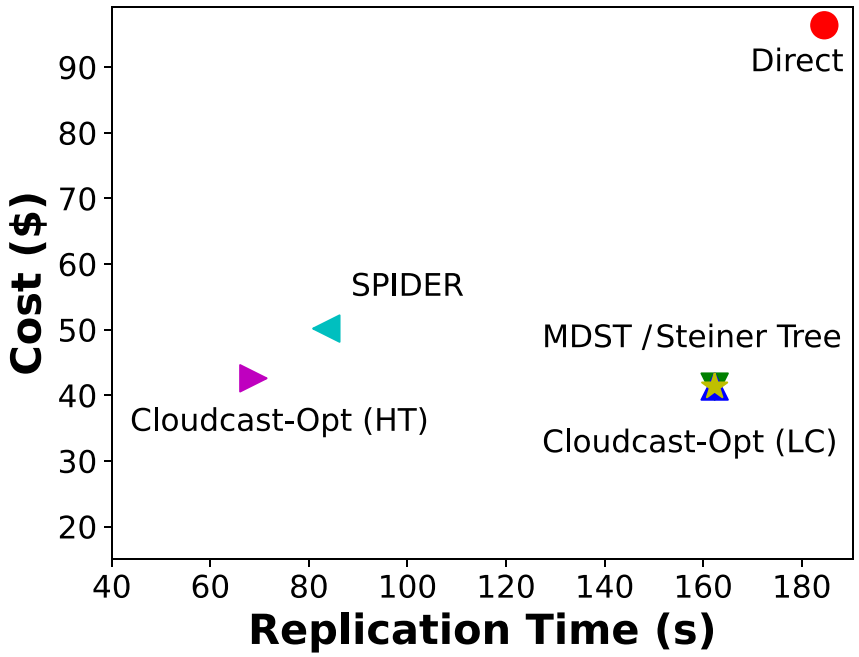


Figure 3.10: Azure Intra-Cloud

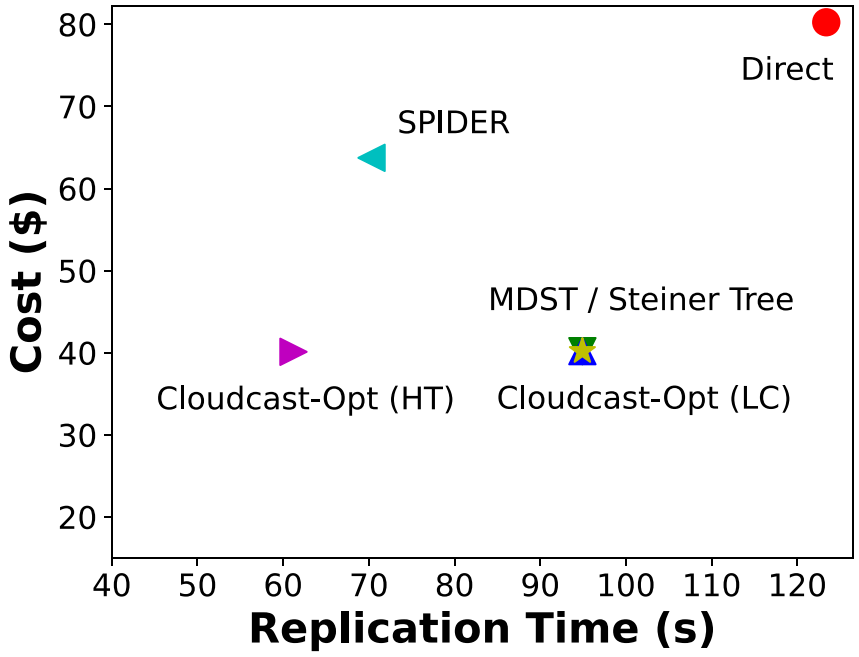


Figure 3.11: GCP Intra-Cloud

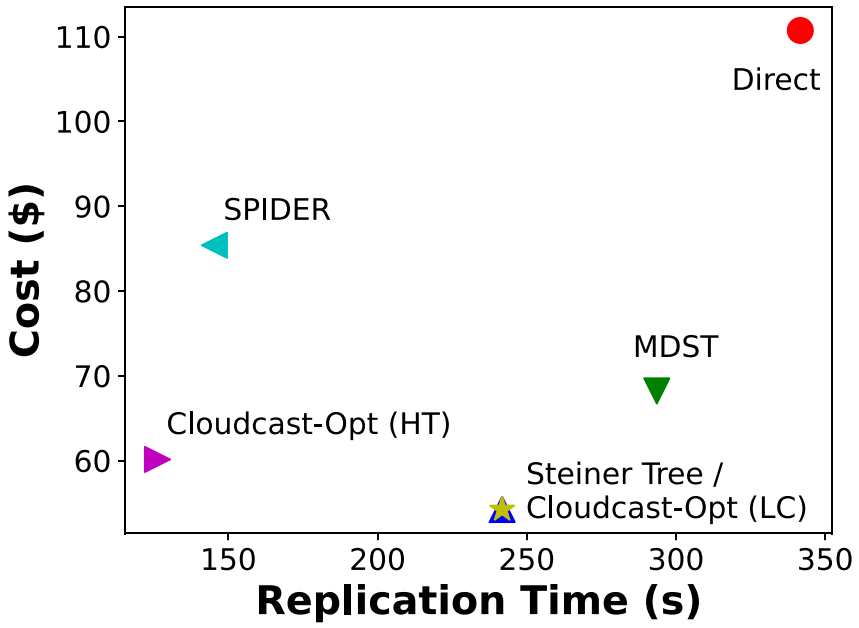


Figure 3.12: Inter-cloud multicast results for different algorithms implemented on Cloudcast. The Cloudcast replication tree is visualized in Figure 3.6.

3.5.2 Cloud Provider and P2P Systems

We run end-to-end evaluation comparing Cloudcast with a commercial baseline (AWS S3 multi-region bucket replication) and P2P systems (BitTorrent and Bullet).

AWS S3 Multi-Region bucket replication

We run an end-to-end comparison between Cloudcast and AWS's S3 multi-region bucket replication AWS Cross-Region Replication (2023) for single-provider multicast. AWS supports adding multiple replication rules to a source bucket to specify automatic replication to one or more replication buckets. In the aspect of time control, AWS supports a replication time control with a minimum 15-minute SLO. However, we found that in our experiments, replications typically completed much faster than 15 minutes. Therefore, we use the actual replication time as a point of comparison.

We compare AWS's replication time and cost to Cloudcast with the planner implemented with both direct transfer and the optimizer. We transfer an OPT model Zhang et al. (2022) with 66 billion parameters (122 GB in total across 9 files) between regions in a single continent[6]. To evaluate AWS replication time and cost, we create buckets with replication rules from a bucket in the source region to buckets in destination regions. Once the replication rules are created, we copy data from a bucket in the same region into the source bucket with 16 VMs. After the write completes, we measure the time until the completion of replication into all destination buckets. We calculate the transfer cost according to AWS's pricing

[6]from `aws:ap-east-1` to `aws:ap-southeast-2`, `aws:ap-south-1`, `aws:ap-northeast-3`, `aws:ap-northeast-2`, `aws:ap-northeast-1`

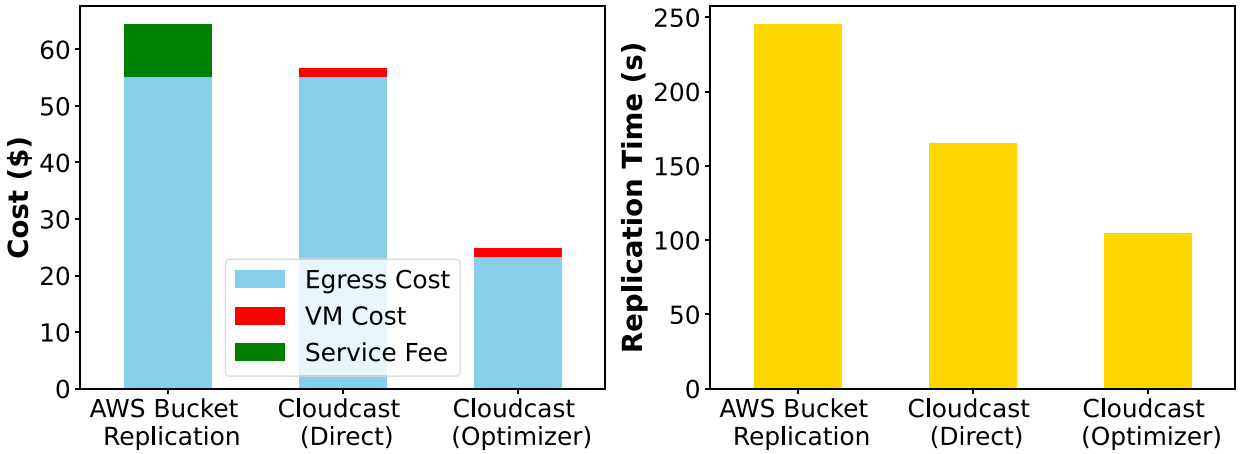


Figure 3.13: Cloudcast outperforms AWS S3 Replication Time Control while reducing total transfer costs.

page Overview of Data Transfer Costs for Common Architectures (2023). We compare AWS multi-region bucket replication to Cloudcast implemented with both the direct and optimizer planner and running. As shown in Figure 3.13, the direct transfer has the same egress costs as AWS bucket replication, but the VM costs are much less than the service fee charged by AWS for the replication. Overall, Cloudcast with the optimizer is able to achieve 2.3× replication speedup and 61.5% cost savings. This is a result of being able to leverage VM parallelism as well as an overlay network that minimizes total egress costs.

P2P BitTorrent and Bullet

We also compare Cloudcast against P2P systems like BitTorrent and Bullet. We run the same transfer benchmark in Azure in Figure 3.10, sending 100GB within Azure to 6 destination regions. We host our own BitTorrent tracker and use aria2 Tsujikawa & Maier (2008) as a BitTorrent client. Since Bullet's implementation is not available, we evaluate Bullet by implementing Bullet's algorithm inside Cloudcast's planner. The result is shown in Figure 3.14: both BitTorrent and Bullet have lower egress costs than direct but higher than Cloudcast. BitTorrent is the slowest because most clients cannot utilize the full bandwidth. The clients are built for scenarios like background seeding and transfer off the critical path, rather than for bulk data transfer. Interestingly, without a centralized planner, BitTorrent is able to find a low-cost multicast replication tree by inferring the bandwidth among peers and preferring the data from peers who have the highest throughput. However, it is still significantly more expensive than Cloudcast.

3.5.3 Ablations of Cloudcast's Optimizer

To understand how our optimizer behaves for different selections of source and destination regions and different target replication times, we run simulated ablations.

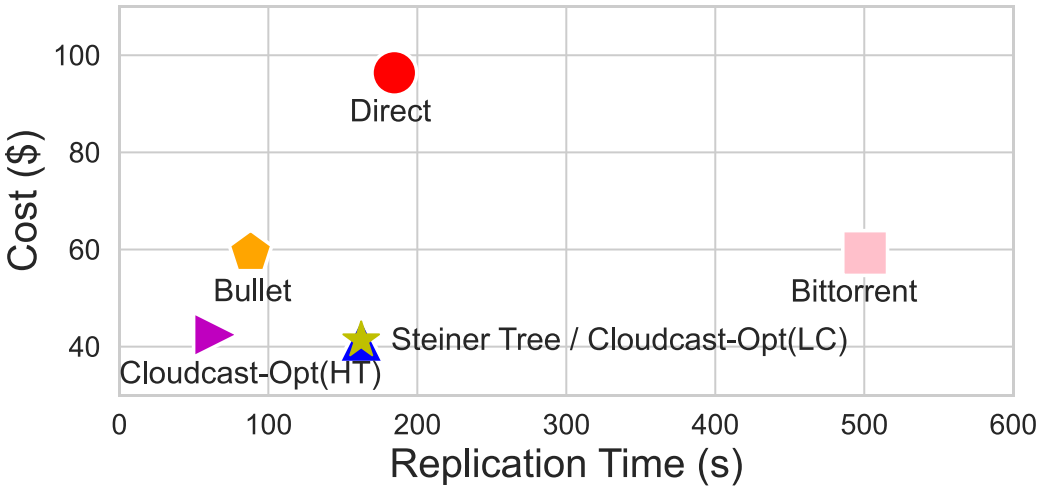


Figure 3.14: Comparison with BitTorrent protocol on the intra-cloud Azure workload in Figure 3.10.

Varying region selection

We test the generality of our improvements by randomly selecting source and destination regions for varying numbers of destinations. We show aggregated results over 100 samples for different numbers of destinations in Figure 3.15. Cloudcast is able to improve the runtime and cost of replication consistently across varying numbers of destinations. Cost and throughput improvement increase with more destinations, since more destinations provide a larger optimization space.

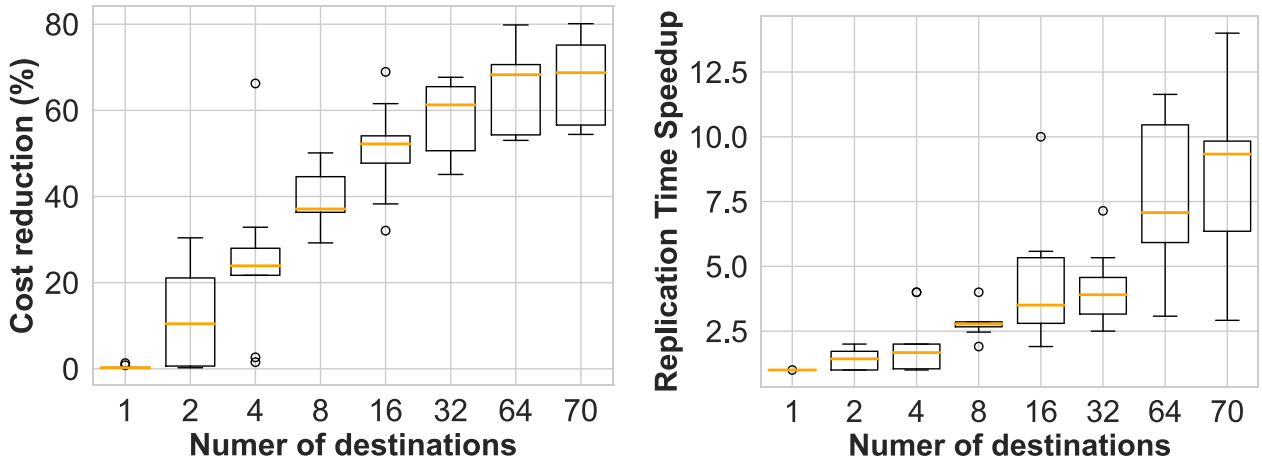
Impact of approximations on solutions

We evaluate how the optimizer with and without approximations scales to larger numbers of destinations in Figure 3.16, by randomly selecting source and destination regions for varying numbers of destination regions. We find that combining all three approximation mechanisms is necessary to scale the optimizer: using no approximations, or only one approximation, takes several minutes for just 10 destinations while using all approximations together reduces solve time to seconds.

We also evaluate how approximations affect the quality of the solution using the monetary cost of the solver-generated solution. We randomly sample 100 source/destination combinations for 5 destinations and compute the difference in the solution's monetary cost and replication runtime compared to MILP without approximation in Table 3.3. We find that the difference in cost averages around 1%, and estimate the worst-case approximation ratio to be 1.4. We find that for even just 5 destinations, the approximated solver runs with a geometric-mean speedup of 30.68×.

Accuracy of replication time model

We compare optimizer-modeled throughput and real throughput in Table 3.4. As transfer size increases, the approximation becomes more accurate. This is because Cloudcast's



(a) Cost Reduction **(b)** Replication Time Speedup

Figure 3.15: Cloudcast optimizer's cost and time improvement over direct replication with varying destination numbers.

Method	Mean error	Solver speedup (geomean)
Node Clustering	0.3%	9.04×
Hop Constraining	1.1%	5.72×
Stripe Iterative	0.0%	7.02×
All Approximations	1.1%	30.68×

Table 3.3: Solve time and solution quality with approximations.

optimizer, designed for bulk data replication, makes several simplifying assumptions, such as perfectly pipelined stripes. Thus, transient inefficiencies during startup and teardown mean smaller transfers may experience lower throughput than the optimizer expects, but for larger, more expensive transfers, modeled throughput closely matches empirical results.

3.5.4 When to Use Cloudcast for Multicast?

Cloudcast is designed for bulk multicast replication in the cloud, so should only be used with data sizes are sufficiently large. Since Cloudcast relies on creating VMs in the cloud at transfer initiation time, there is a constant overhead from VM startup time. We calculate the transfer size break-even point (i.e. the minimum data size for using Cloudcast) for varying providers and VM capacity limits (constraining the throughput for the Cloudcast overlay), shown in Figure 3.17. We approximate the per-destination replication throughput without Cloudcast as equal to the per-VM egress bandwidth limit, ignoring congestion between source and destination VMs. Azure has a higher break-even point than AWS and GCP due to two effects. First, the VM startup time is the highest of all providers (56 seconds). Second, VMs in Azure are not subjected to egress constraints (5 Gbps and 7 Gbps for AWS and GCP, respectively). As a result, the benefits of using Cloudcast's techniques are only realized for

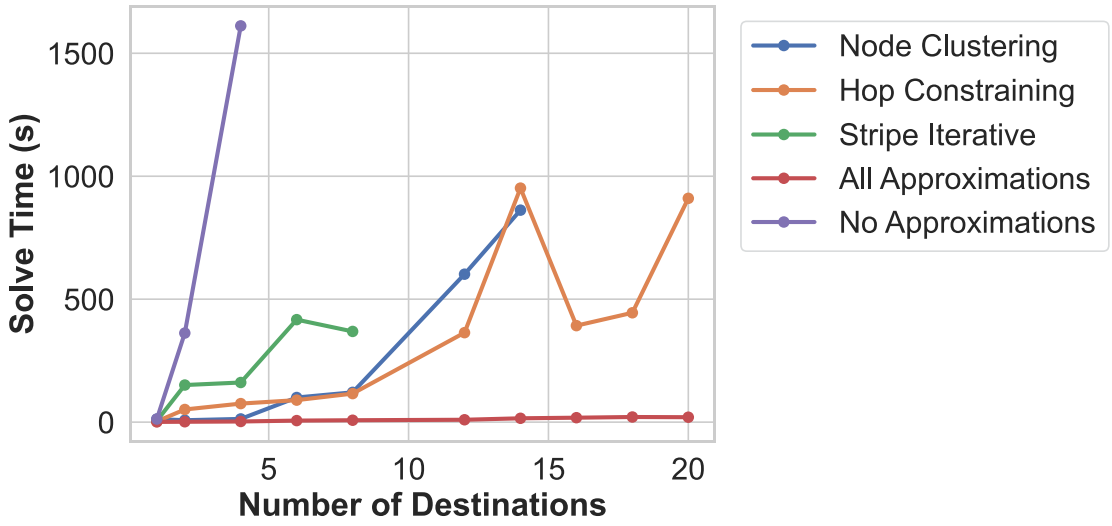


Figure 3.16: Approximations reduce solver runtime from the cutoff of 30 minutes to seconds for up to 20 destinations.

Transfer Size (GB)	Prediction error
16	16.6%
32	8.51%
64	3.31%
128	1.69%

Table 3.4: Accuracy of the optimizer's predicted throughput.

larger transfer sizes or larger numbers of destinations.

3.6 Related Work

Overlay Unicast. A significant body of prior work uses overlay networks to improve the performance and resilience of one-to-one data transfers in the Internet and peer-to-peer networks Andersen et al. (2001); Kostić et al. (2003); Castro et al. (2003). In clouds, previous work has also leveraged cloud elasticity to further improve performance Matos et al. (2009); Jain et al. (2022). However, they do not consider multicast, and except Skyplane Jain et al. (2022), none consider the monetary cost of replication in the cloud. Handling multicast is challenging. For example, while Jain et al. (2022) can leverage elastic resources, cloud pricing models, and overlay networking for bulk unicast replication in the cloud, its techniques are not directly applicable to the multicast setting. More specifically, Skyplane's flow-based throughput model results in ambiguous multicast distribution tree solutions as it ignores the identity of data sent along multiple paths. Furthermore, since Skyplane's optimizer is not designed for multicast, it cannot take advantage of techniques such as leveraging multiple distribution trees to alleviate source bottlenecks.

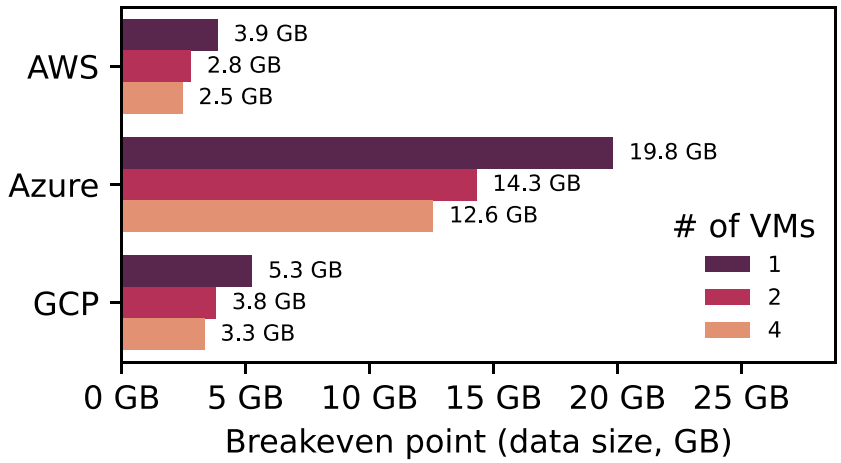


Figure 3.17: Estimated break-even point for a 6-destination replication based on VM startup times (35, 56, and 34 seconds for AWS, Azure, and GCP, respectively) and VM egress limits.

Overlay Multicast. End-system multicast Chu et al. (2002) and overlay multicast have been proposed to efficiently disseminate data from a single source to multiple destinations. Many application-level multicast algorithms have been proposed. Algorithms like SPIDER Ganguly et al. (2005), SplitStream Castro et al. (2003), Bullet Kostić et al. (2003), and Overcast Jannotti et al. (2000) are designed for high-bandwidth, cross-internet file distribution with application-level multicast overlays. However, like with most overlay unicast systems, these algorithms ignore monetary costs and focus on techniques to maximize bandwidth.

Inter-DC Replication. Extensive prior work addresses inter-DC replication Sima et al. (2022); Flinn et al. (2022); Zhang et al. (2018); Laoutaris et al. (2011); Feng et al. (2012), including bulk multicast Luo et al. (2019). Recent research includes deadline Luo et al. (2019) and cost-awareness Fatemipour et al. (2022); Feng et al. (2012). However, the cost model in the Inter-DC setting cannot be easily adapted to cloud users, for which network pricing is based on total data volume rather than bandwidth. Furthermore, existing formulations are not designed for multicast Fatemipour et al. (2022); Feng et al. (2012) or do not consider more than a few geo-distributed regions Luo et al. (2019). Our work focuses on public clouds, considering unique per-GB network pricing, elastic resources, and cloud-specific resource constraints. Our approximation algorithm is also designed to scale to all regions across multiple cloud vendors.

Traffic Engineering. The classic problem of traffic engineering has also formulated optimization problems for minimizing cost under performance constraints. These techniques have recently been applied to cloud providers and their monetary costs. For example, Entact Zhang et al. (2010) studied how to optimize costs for online service providers while still minimizing user latency. Similarly, Cascara Singh et al. (2021) leveraged latency-equivalent

paths to identify cost-minimizing paths for cloud providers. Like Inter-DC Replication, these approaches have been developed from the perspective of the cloud or service provider and, thus, present a materially different optimization problem.

Steiner Trees. The Steiner Tree algorithm has been applied in the multicast setting both to minimize costs in terms of delay Jiang & Chen (2016) and cloud egress costs García-Dorado & Rao (2015)). CloudMPCast García-Dorado & Rao (2015) minimizes egress costs in cloud bulk data multicast by constructing a Steiner Tree overlay network that avoids low-throughput cross-region paths. However, CloudMPCast overlooks VM capacity and per-VM egress/ingress limits in its MILP formulation. Also, CloudMPCast aims to achieve *comparable* performance to direct transfers while minimizing cost, unlike Cloudcast, which optimizes throughput.

Geo-Distributed Storage. Geo-distributed storage via data replication is supported by a variety of cloud services, such as AWS Cross-Region Replication AWS Cross-Region Replication (2023), AWS Multi-Region Access Points Casalboni (2021), and GCP Multi-region buckets GCP Multi-Region bucket (2022). Cross-region replicated buckets (e.g., S3 replication rules) automatically replicate written data from a bucket in one region to one or more buckets in other regions. However, these services have limited support for cross-cloud data movement and do not minimize egress costs even for intra-cloud data movement. SPANStore Wu et al. (2013) designs a system for geo-distributed storage across multiple cloud providers, and also optimizes egress costs of relaying data on PUT requests. However, its relay strategy is optimized for latency, not bandwidth.

Peer-to-peer Multicast. Peer-to-peer systems (P2P) support file sharing among a set of end-user clients. The BitTorrent protocol Cohen (2003) reduces the network load on the source by allowing clients to upload and download data to each other. BitTorrent is widely used for data multicast in data center environments by Facebook and Twitter TorrentFreak (2023). Specialized systems for data multicast that use BitTorrent include Uber's Kraken Uber (2022) and Ant Group's Dragonfly Alibaba (2018). These P2P systems have significant overhead as they are designed for adversarial settings where peers may be unreliable or fail. Moreover, P2P systems must scale to millions of destinations and therefore lack centralized control which prevents custom routing topologies. P2P systems may redundantly send data over expensive links due to a lack of cost awareness.

Method	Multicast	Cloud Pricing	Striping	Resource Elasticity
Unicast overlay networks				
RON Andersen et al. (2001)	×	×	✓	×
Skyplane Jain et al. (2022)	×	✓	✓	✓
COMS Fatemipour et al. (2022)	×	×	×	∼
Peer-to-peer				
BitTorrent Cohen (2003)	✓	×	✓	×
SplitStream Castro et al. (2003)	✓	×	✓	×
Bullet Kostić et al. (2003)	✓	×	✓	×
Inter-DC overlay multicast				
SPIDER Ganguly et al. (2005)	✓	×	✓	×
CodedBulk Tseng et al. (2021)	✓	×	✓	×
BDS Zhang et al. (2018)	✓	×	✓	×
Deadline-aware Inter-DC Ji et al. (2018)	✓	×	×	✓
Cost optimized overlay networks				
SPANStore Wu et al. (2013)	✓	✓	×	×
CloudMPCast García-Dorado & Rao (2015)	✓	✓	×	×
Jetway Feng et al. (2012)	×	✓	✓	×
Cloudcast (ours)	✓	✓	✓	✓

Table 3.5: Cloudcast builds on prior work by enabling multicast, optimizing cloud costs, and leveraging cloud resource elasticity and multiple distribution trees.

Chapter 4

Conclusion & Future Work

The increasing complexity and distribution of modern machine learning systems have brought the challenge of maintaining data freshness to the forefront of ML infrastructure design. This thesis has presented a comprehensive approach to ensuring data freshness across clouds for model serving, addressing both the computational aspects of feature maintenance and the networking challenges of efficient data transfer.

In this work, 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.

We also explored the problem of cost-optimized cloud multicast by introducing overlay networks of ephemeral VM waypoints that exploit path-specific cloud pricing to significantly reduce cost and improve throughput. We developed a MILP formulation of this problem and introduced approximations that make the solving time feasible for practical applications. Our evaluation against academic and commercial baselines demonstrated up to a 61.5% reduction in cost and a 2.3× improvement in runtime. Cloudcast has been released as part of the Skyplane open source project with pluggable planning algorithms to enable future research in this space.

Collectively, these systems represent a significant step forward in our ability to ensure data freshness for model serving in multi-cloud environments. The impact of these contributions

extends beyond just improving prediction accuracy or reducing computational costs. By enabling more efficient maintenance of fresh data across distributed environments, this work supports the development of more responsive, adaptive, and reliable ML systems. It allows organizations to more effectively leverage multi-cloud strategies, enables more frequent model updates, and supports the deployment of ML applications in diverse geographic regions while maintaining consistent performance. In conclusion, as machine learning continues to permeate critical applications and services, the importance of maintaining fresh, accurate data will only grow. The systems and techniques presented in this thesis provide a solid foundation for addressing these challenges, paving the way for more efficient, effective, and reliable distributed machine learning infrastructure in the cloud era.

Bibliography

2021, CVXPY: A Python Library for Convex Optimization
2023, LZ4 - Extremely fast compression
2023, PyNaCl: Python binding to the libsodium library
Abadi, M., Barham, P., Chen, J., et al. 2016, in 12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16), 265
Agarwal, S., Mozafari, B., Panda, A., et al. 2013, in Proceedings of the 8th ACM European Conference on Computer Systems, 29
Alibaba. 2018, Dragonfly, https://github.com/dragonflyoss/Dragonfly, accessed on 12/15/2022
Alteryx. 2023, Feature Tools
Andersen, D., Balakrishnan, H., Kaashoek, F., & Morris, R. 2001, in Proceedings of the eighteenth ACM symposium on Operating systems principles, 131
AWS, A. 2023, EC2 On-Demand Instance Pricing, https://aws.amazon.com/ec2/pricing/on-demand
AWS Cross-Region Replication. 2023, AWS Cross-Region Replication, https://docs.aws.amazon.com/AmazonS3/latest/userguide/replication.html
Azure, M. 2023, Pricing - Bandwidth, https://azure.microsoft.com/en-us/pricing/details/bandwidth/
Bailis, P., Fekete, A., Franklin, M. J., et al. 2014, Proc. VLDB Endow., 8, 185–196
Bailis, P., Venkataraman, S., Franklin, M. J., Hellerstein, J. M., & Stoica, I. 2012, Proc. VLDB Endow., 5, 776–787
Bhardwaj, R., Xia, Z., Ananthanarayanan, G., et al. 2020, arXiv preprint arXiv:2012.10557
Casalboni, A. 2021, Amazon S3 Multi-Region Access Points, https://aws.amazon.com/s3/features/multi-region-access-points/
Castro, M., Druschel, P., Kermarrec, A.-M., et al. 2003, ACM SIGOPS operating systems review, 37, 298
Chasins, S., Cheung, A., Crooks, N., et al. 2022, arXiv preprint arXiv:2205.07147
Chaudhuri, S., Ding, B., & Kandula, S. 2017, in Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD '17 (New York, NY, USA: Association for Computing Machinery), 511–519
Chirkova, R., Yang, J., et al. 2011, Foundations and Trends in Databases, 4, 295
Chu, Y., Rao, S., Seshan, S., & Zhang, H. 2001, in Proceedings of the 2001 conference on

Applications, technologies, architectures, and protocols for computer communications, 55
Chu, Y.-h., Rao, S. G., Seshan, S., & Zhang, H. 2002, IEEE Journal on selected areas in communications, 20, 1456
Cloud, G. 2023, All networking pricing, https://cloud.google.com/vpc/network-pricing
Cohen, B. 2003, in Workshop on Economics of Peer-to-Peer systems, Vol. 6, Berkeley, CA, USA, 68
Cohere. 2023, Scalable, affordable pricing
Cooper, B. F., Ramakrishnan, R., Srivastava, U., et al. 2008, Proceedings of the VLDB Endowment, 1, 1277
Cortez, E., Bonde, A., Muzio, A., et al. 2017, in Proceedings of the 26th Symposium on Operating Systems Principles, 153
Crankshaw, D., Sela, G.-E., Mo, X., et al. 2020, in Proceedings of the 11th ACM Symposium on Cloud Computing, 477
Crankshaw, D., Wang, X., Zhou, G., et al. 2017, in 14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17), 613
Crooks, N., Pu, Y., Estrada, N., et al. 2016, in Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16 (New York, NY, USA: Association for Computing Machinery), 1615–1628
Cui, H., Cipar, J., Ho, Q., et al. 2014, in Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference, USENIX ATC'14 (USA: USENIX Association), 37–48
Denis, F. 2013, The Sodium cryptography library
Fatemipour, B., Shi, W., & St-Hilaire, M. 2022, in 2022 IEEE Cloud Summit, IEEE, 17
Feng, Y., Li, B., & Li, B. 2012, in Proceedings of the 20th ACM international conference on Multimedia, 259
Flinn, J., Dou, X., Aggarwal, A., et al. 2022, in 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), 1
Ganguly, S., Saxena, A., Bhatnagar, S., Izmailov, R., & Banerjee, S. 2005, in Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies., Vol. 4, IEEE, 2246
García-Dorado, J. L., & Rao, S. G. 2015, IEEE Transactions on Cloud Computing, 7, 34
GCP Multi-Region bucket. 2022, GCP Multi-Region bucket, https://cloud.google.com/storage/docs/locations#location-mr, accessed on 12/15/2022
Gjengset, J., Schwarzkopf, M., Behrens, J., et al. 2018, in 13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18), 213
Grouplens. 2023, MovieLens 1M Dataset
Gurobi Optimization, LLC. 2023, Gurobi Optimizer Reference Manual
Guu, K., Lee, K., Tung, Z., Pasupat, P., & Chang, M. 2020, in International conference on machine learning, PMLR, 3929
Hopsworks. 2023a, Feature Stores Org
—. 2023b, Hopsworks
Hwang, F. K., & Richards, D. S. 1992, Networks, 22, 55

Jain, P., Kumar, S., Wooders, S., et al. 2022, arXiv preprint arXiv:2210.07259
Jannotti, J., Gifford, D. K., Johnson, K. L., Kaashoek, M. F., & O'Toole Jr, J. W. 2000, in Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)
Ji, S., Liu, S., & Li, B. 2018, in 2018 IEEE International Conference on Cloud Engineering (IC2E), 124
Jiang, J., Ananthanarayanan, G., Bodik, P., Sen, S., & Stoica, I. 2018, in Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, 253
Jiang, J.-R., & Chen, S.-Y. 2016, in Proceedings of the 11th International Conference on Future Internet Technologies, 1
Kakantousis, T., Kouzoupis, A., Buso, F., et al. 2019, in Proc. 2nd SysML Conf., Palo Alto, USA
Kang, D., Emmons, J., Abuzaid, F., Bailis, P., & Zaharia, M. 2017, arXiv preprint arXiv:1703.02529
Kang, D., Gan, E., Bailis, P., Hashimoto, T., & Zaharia, M. 2020, arXiv preprint arXiv:2004.00827
Kohler, E., Morris, R., Chen, B., Jannotti, J., & Kaashoek, M. F. 2000, ACM Transactions on Computer Systems (TOCS), 18, 263
Koren, Y., Bell, R., & Volinsky, C. 2009, Computer, 42, 30
Kostić, D., Rodriguez, A., Albrecht, J., & Vahdat, A. 2003, in Proceedings of the nineteenth ACM symposium on Operating systems principles, 282
Kraft, P., Kang, D., Narayanan, D., et al. 2019, arXiv preprint arXiv:1906.01974
Kraska, T., Pang, G., Franklin, M. J., Madden, S., & Fekete, A. 2013, in Proceedings of the 8th ACM European Conference on Computer Systems, EuroSys '13 (New York, NY, USA: Association for Computing Machinery), 113–126
Laoutaris, N., Sirivianos, M., Yang, X., & Rodriguez, P. 2011, in Proceedings of the ACM SIGCOMM 2011 Conference, 74
Laptev, N., & Amizadeh, S. 2015, http://webscope.sandbox.yahoo.com/catalog.php
Lee, K., Chang, M.-W., & Toutanova, K. 2019, arXiv preprint arXiv:1906.00300
Lewis, P., Perez, E., Piktus, A., et al. 2020, Advances in Neural Information Processing Systems, 33, 9459
Li, C., Porto, D., Clement, A., et al. 2012, in Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12 (USA: USENIX Association), 265–278
Li, M., Wang, Y.-X., & Ramanan, D. 2020, in European Conference on Computer Vision, Springer, 473
Lloyd, W., Freedman, M. J., Kaminsky, M., & Andersen, D. G. 2011, in Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP '11 (New York, NY, USA: Association for Computing Machinery), 401–416
Luo, L., Jin, Q., Xie, J., Sun, G., & Yu, H. 2021, IEEE Transactions on Services Computing
Luo, L., Kong, Y., Noormohammadpour, M., et al. 2019, IEEE Transactions on Cloud Computing, 10, 304
Matos, M., Sousa, A., Pereira, J., & Oliveira, R. 2009, in Proceedings of the Third Workshop

on Dependable Distributed Data Management, 14
McSherry, F., Lattuada, A., Schwarzkopf, M., & Roscoe, T. 2020, Proc. VLDB Endow., 13, 1793
Mishra, A., Sriharsha, R., & Zhong, S. 2021, arXiv preprint arXiv:2107.09110
Moritz, P., Nishihara, R., Wang, S., et al. 2018, in 13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18), 561
Murray, D. G., McSherry, F., Isaacs, R., et al. 2013, in Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP), 439
Naumov, M., Mudigere, D., Shi, H. M., et al. 2019, CoRR, abs/1906.00091
OpenAI. 2023, OpenAI, https://platform.openai.com/docs/guides/rate-limits?context=tier-free, last accessed November 11, 2023
Overview of Data Transfer Costs for Common Architectures. 2023, Overview of Data Transfer Costs for Common Architectures, https://aws.amazon.com/blogs/architecture/overview-of-data-transfer-costs-for-common-architectures/, accessed on 12/15/2022
Packer, C., Fang, V., Patil, S. G., et al. 2023, arXiv preprint arXiv:2310.08560
Peter, S., Javed, U., Zhang, Q., et al. 2014, ACM SIGCOMM Computer Communication Review, 44, 99
Prince, M., & Rao, N. 2021, AWS's Egregious Egress, https://blog.cloudflare.com/aws-egregious-egress/
Redis. 2023, Redis
Rehfeldt, D., & Koch, T. 2021, in Lecture Notes in Computer Science, Vol. 12707, Integer Programming and Combinatorial Optimization - 22nd International Conference, IPCO 2021, Atlanta, GA, USA, May 19-21, 2021, Proceedings, ed. M. Singh & D. P. Williamson (Springer), 473
Seabold, S., & Perktold, J. 2010, in 9th Python in Science Conference
Services, A. W. 2023, Amazon SageMaker Feature Store
Sima, C., Fu, Y., Sit, M.-K., et al. 2022, in 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), 821
Singh, R., Agarwal, S., Calder, M., & Bahl, P. 2021, in 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21), 201
Sivasubramanian, S. 2012, in Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, 729
Stoica, I., Adkins, D., Zhuang, S., Shenker, S., & Surana, S. 2002, in Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, 73
Tecton. 2023, Tecton
Terry, D. B., Prabhakaran, V., Kotla, R., et al. 2013, in Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13 (New York, NY, USA: Association for Computing Machinery), 309–324
TorrentFreak. 2023, Facebook Uses BitTorrent, and They Love It, https://torrentfreak.com/facebook-uses-bittorrent-and-they-love-it-100625/, accessed

on 12/15/2022
Tseng, S.-H., Agarwal, S., Agarwal, R., Ballani, H., & Tang, A. 2021, in NSDI, 15
Tsujikawa, T., & Maier, N. 2008, aria2 - The ultra fast download utility, https://github.com/aria2/aria2
Uber. 2022, P2P Docker registry capable of distributing TBs of data in seconds, https://github.com/uber/kraken, accessed on 12/15/2022
Villalba, M. 2020, Amazon S3 Replication Adds Support for Multiple Destination Buckets, https://aws.amazon.com/blogs/aws/new-amazon-s3-replication-adds-support-for-multiple-destination-buckets/
Wang, Z., Lin, X., Mishra, A., & Sriharsha, R. 2021, in 2021 International Conference on Data Mining Workshops (ICDMW), IEEE, 414
Wong, M. H., & Agrawal, D. 1992, in Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS '92 (New York, NY, USA: Association for Computing Machinery), 236–245
Wu, K.-L., Yu, P., & Pu, C. 1992, in [1992] Eighth International Conference on Data Engineering, 506
Wu, Z., Butkiewicz, M., Perkins, D., Katz-Bassett, E., & Madhyastha, H. V. 2013, in Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, 292
Yang, Z., Wu, Z., Luo, M., et al. 2023, in 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI '23)
Yu, H., & Vahdat, A. 2000, in Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4, OSDI'00 (USA: USENIX Association)
Zhang, S., Roller, S., Goyal, N., et al. 2022, OPT: Open Pre-trained Transformer Language Models, arXiv:2205.01068 [cs.CL]
Zhang, Y., Jiang, J., Xu, K., et al. 2018, in Proceedings of the Thirteenth EuroSys Conference, 1
Zhang, Z., Zhang, M., Greenberg, A. G., et al. 2010, in NSDI, 33
Zhou, J., Larson, P.-k., & Goldstein, J. 2005, in submitted to this conference