CacheW

Machine Learning Input Data Processing as a Service

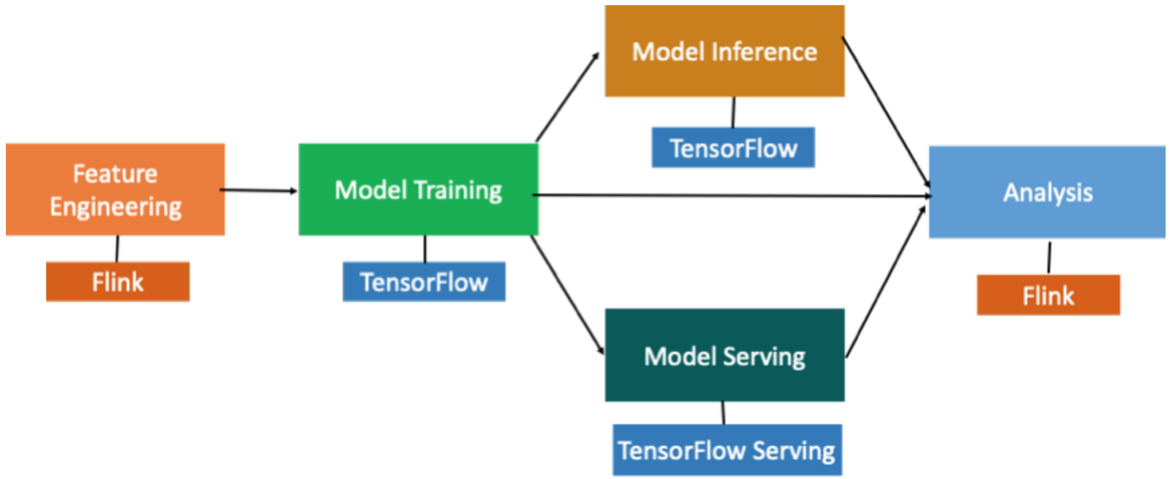
USENIX ATC'22

Dan Graur, Damien Aymon, Dan Kluser, and Tanguy Albrici, ETH Zurich;Chandramohan A. Thekkath, Google; Ana Klimovic, ETH Zurich

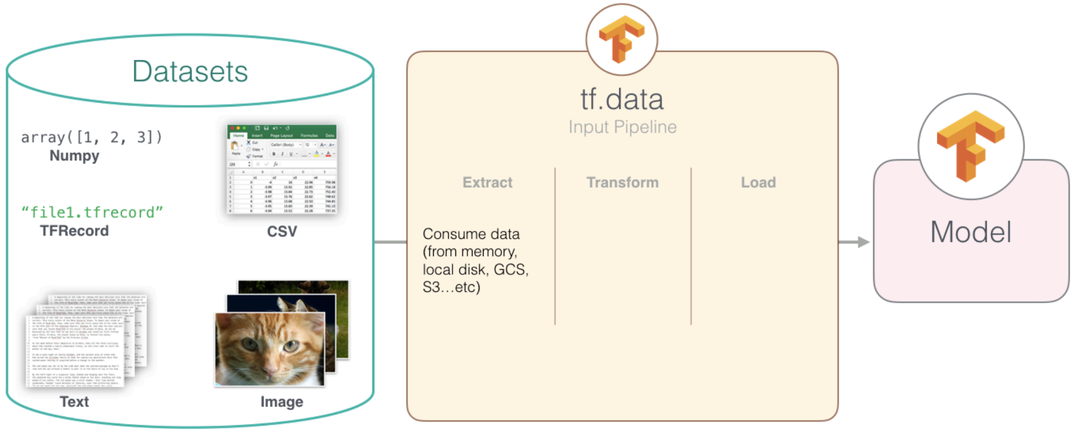
Introduction

Processing input data plays a vital role in ML training

Input pipeline, which is responsible for feeding data-hungry GPUs/TPUs with training examples, **is a common bottleneck**



Feature engineering



Input pipeline

Introduction

Present challenges:

- ML schedulers focus on GPU/TPU resources, leaving users on their own to optimize multi-dimensional resource allocations for data processing **(Lack of focus on data processing)**

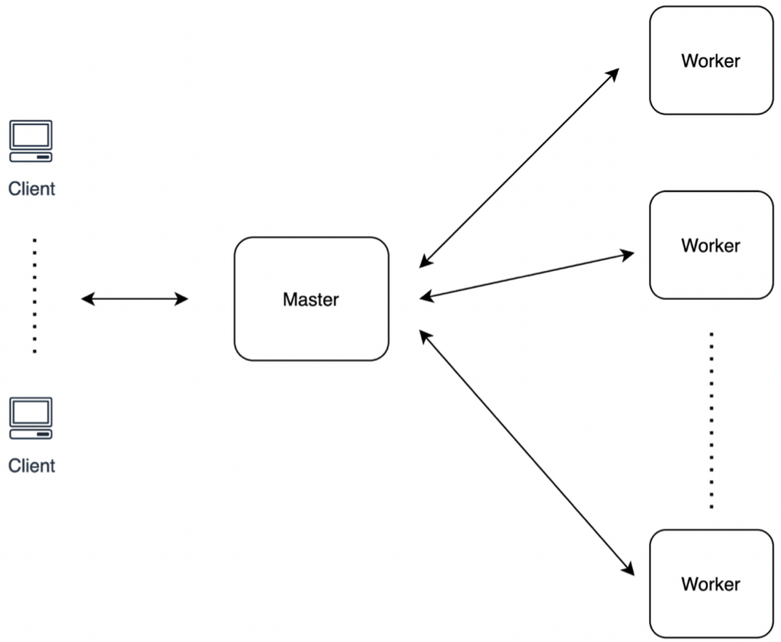
- input pipelines often consume excessive compute power to repeatedly transform the same data **(useless excessive computation for repeated data)**

How to alleviate data stall

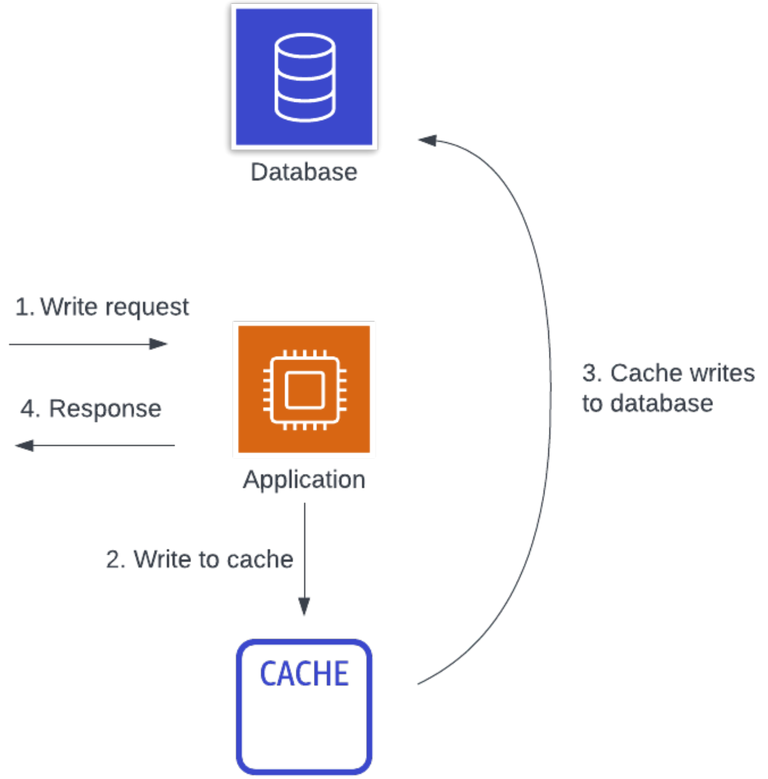
Maximize input pipeline throughput

Introduction

Present attempt



Distribute data processing on remote **CPU** workers



Reuse cached data transformations

Introduction

Problem in cache ?

- Expensive store cost
- Time saved by cache is not always bottleneck
- Reuse transformed data may impact accuracy

Which source or transformed data to cache

Introduction

Problem in distribution ?

cache

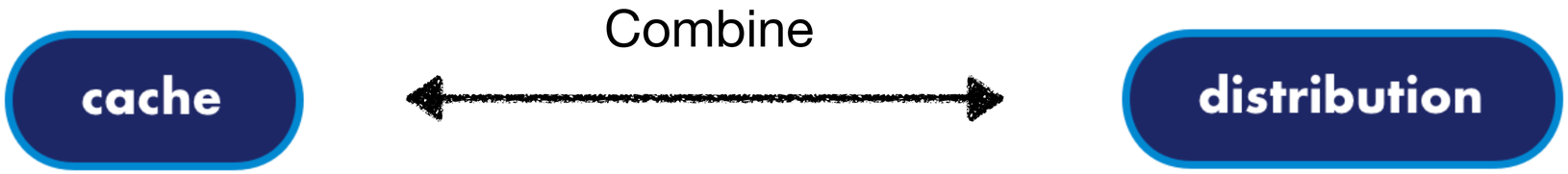
distribution

- Always CPU?
- Way to dynamically scales workers

How to improve parallelism

Introduction

Paper's solution



Which source or transformed data to cache

How to improve parallelism

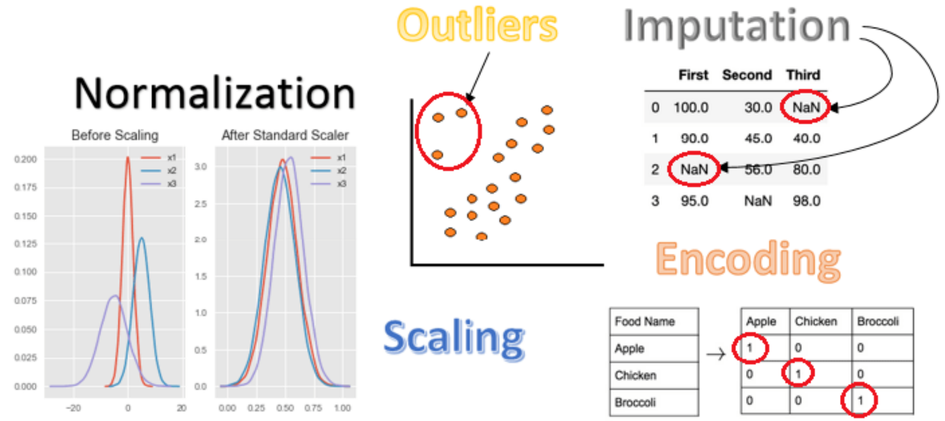
Jointly optimization

Introduction

Importance of input data processing

Transformation—key to achieving high accuracy & low cost & training time

- Extract features **High accuracy**
- Sampling data from imbalanced classes
- Randomly augmenting data



- Ingest data speed from storage **Low cost & training time**
- Apply transformation on-the-fly
- Load transformed data to training nodes

Data Preprocessing

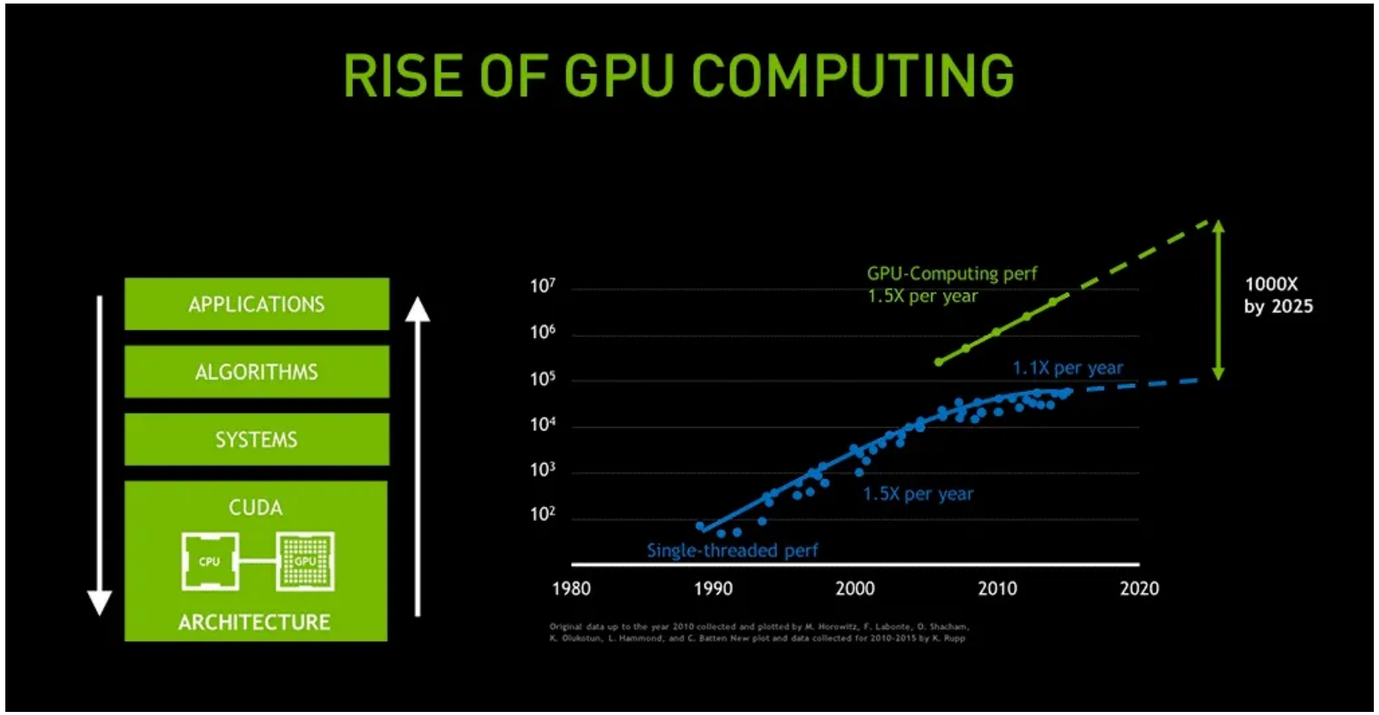
Data processing

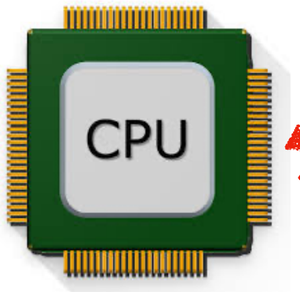
Introduction

Why always CPU ?

1.**CPU's universality** among various ops during input data processing

2.Usage of GPU may cause **high training time**





Responsible for input data processing ✓

Poor performance on ML compared with specialized hardware ✗

Introduction

Disaggregation is essential in ML

Challenge & solution 1 on ML data processing optimization

- Difficulty on allocating **right** amount of
  - CPU
  - Memory
  - Storage
- **Difference on ratio of CPUs** for data processing **and GPUs or TPUs** for training among jobs

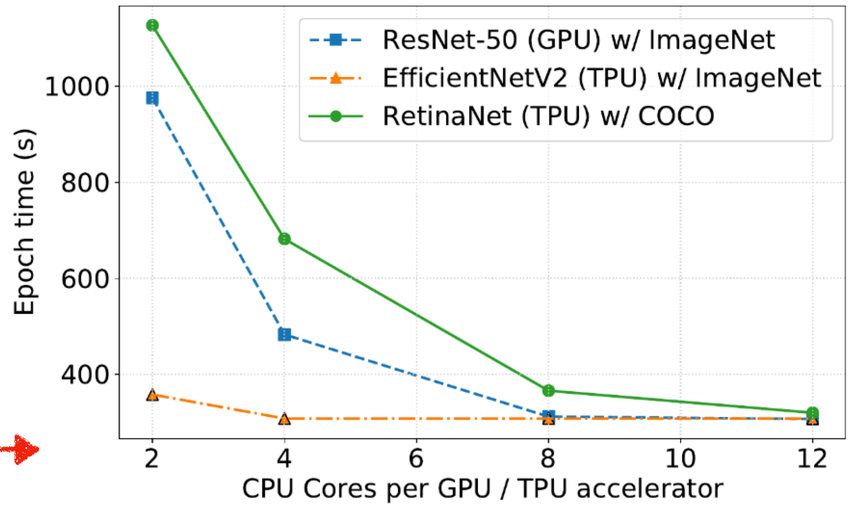


Figure 1: Training jobs benefit differently when given more CPU resources for input data processing per accelerator core.

Solution -> Disaggregation

User's work — Lack of automation

Introduction

More compute & memory resources are required for data processing

Challenge & solution 2 on ML data processing optimization

- data ingestion can consume more power than model training itself

- Despite with frameworks who provide mechanisms for reusing memoized data transformations
- Cache in tf.data & snapshot op TensorFlow
- Hard to determine which (transformed) datasets to cache

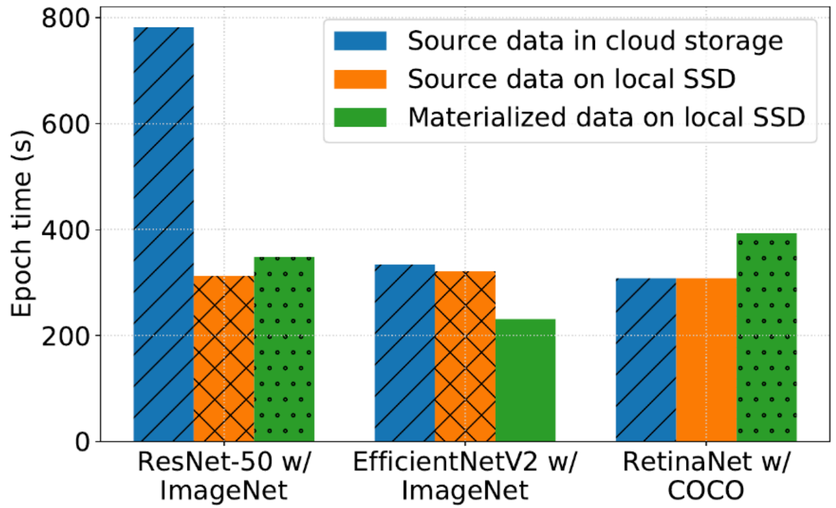


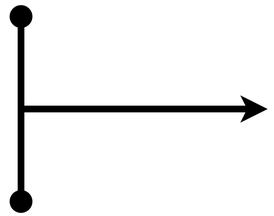
Figure 2: Caching source data or materializing data in local storage does not always improve training throughput.

Introduction

More compute & memory resources are required for data processing

Challenge & solution 2 on ML data processing optimization

- **Distribute** data processing on remote CPU workers
- **Reusing cached data** transformations



Lack of optimization & too rough

- **Autoscale** resource **simply based on CPU & memory utilization**

ML Input Data Processing

ML Input Data Pipeline Characteristics

Four basic steps

- **Reading input data from storage** — low-cost distributed storage & GB~PB in size
- **Transforming data** — raw data -> elements models need
  - basic: decompress data , parse file formats , extract feature , batch elements
  - generalization: randomly sampling , augmenting , shuffling elements

75%

Bigger or smaller → **Vary across jobs**

- **Loading data to training nodes** — autoscaling — to GPUs/TPUs for training

vary dynamically upon hardware FLOPs & algorithm design

- **Re-executing input pipelines** — autocaching — **re-executing is common but useless**

the top 10% most commonly executed input pipelines account for 77% of all executions

Why always CPU — 72% CPU cycles are used for ML input data processing

ML Input Data Processing

Why disaggregate input data processing?

Two major drawbacks

- CPU/memory-intensive input pipelines can **easily saturate host resources and limit training throughput**
- **The ratio of resources** required for input data processing vs model training **varies across jobs**

Two intentions

- Improve **resource utilization**
- **Avoid input data stalls**

Single node

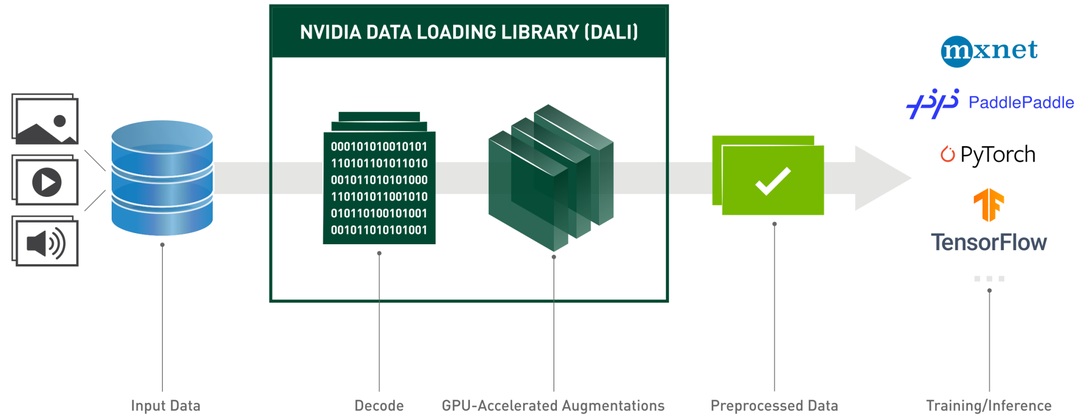
Distributed systems

ML Input Data Processing

Existing Mechanisms

TensorFlow **tf.data**

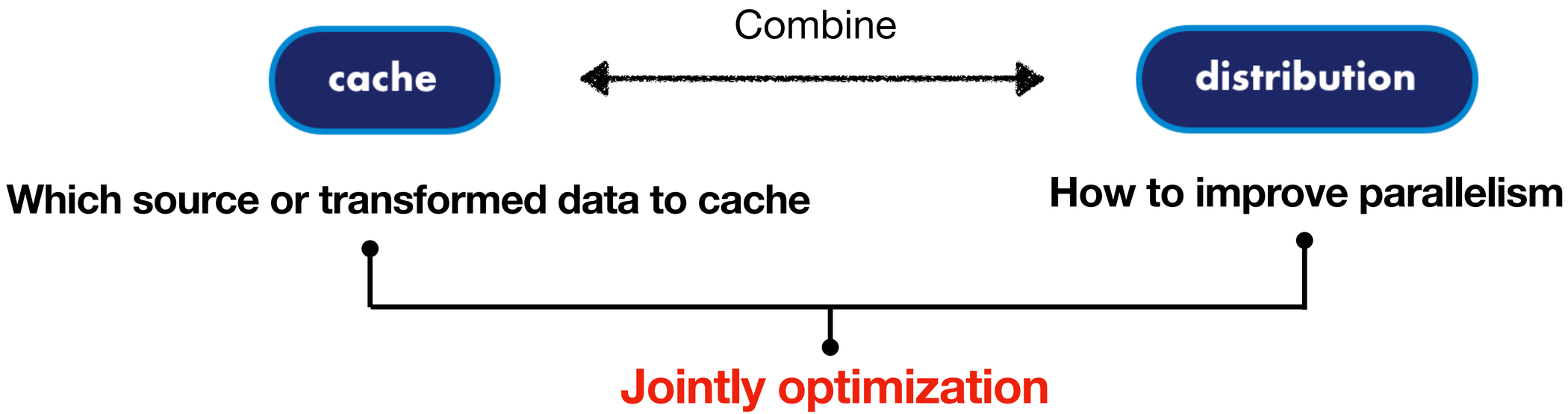
PyTorch



- **Disaggregation** dispatcher (allocate resource) & worker (fetch data)
  user decide 1. Number of input data 2. Per-job resource allocations
- **Dataset Caching** tf.data snapshot op
  user decide where to cache (insert snapshot op)
- **Autotuning** tf.data's runtime & Plumber dynamically tune software parallelism and memory buffer size
  only match single node
- **GPU Offloading** NVIDIA DALI — image data augmentation

ML Input Data Service Challenges

Two key challenges & design of CacheW



autocaching

Save compute resources by **selectively caching (transformed) datasets**

autoscaling

Scale resources (CPU worker) for data processing

ML Input Data Service Challenges

Autoscaling challenges—allocation should be suitable

- **data stalls**, which leave expensive hardware accelerators idle, increasing end-to-end training time and cost **(Too much)**
- **extra costs** without improving end-toend performance **(Too few)**

Autoscaling challenges—how to determine allocation

- each model and input data pipeline combination have **unique requirements (hard to model)**
- training throughput does **not scale linearly** with CPU and memory resources **(difficult to model how resource allocation affects performance)**

Modeling for prediction ✗

ML Input Data Service Challenges

Autocaching challenges—which to cache

- Deciding **which input data transformations to materialize and reuse** versus which transformations to execute online during training is complex
- **Caching does not always improve epoch time**, in particular if reading source data from cloud storage and transforming it on-the-fly is sufficiently fast to saturate model ingestion throughput

Autocaching challenges—cache carefully

- caching and reusing the results of transformations which randomly permute data from one epoch will remove this randomness across epochs
- **reusing random augmentations across epochs is plausible, but must be done sparingly to avoid degrading model accuracy (ATC'2021 Alsys)**

Alsys USENIX Annual Technical Conference (ATC'21), pages 537–550, 2021

CacheW Overview

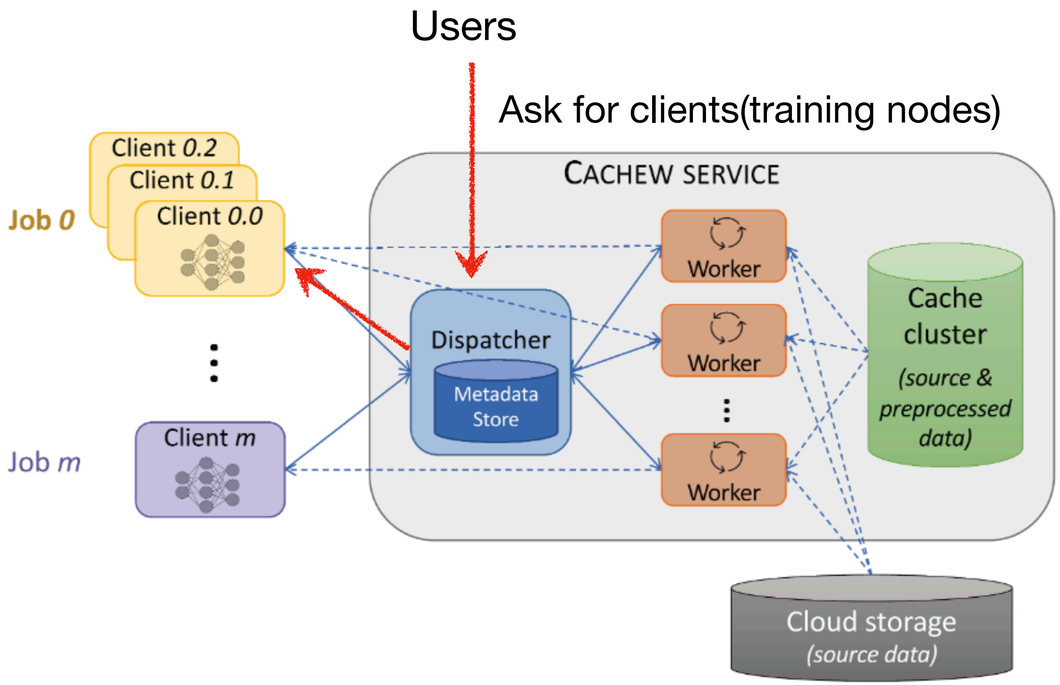


Figure 3: Cachew system architecture. Solid lines depict control logic and metadata communication. Dotted lines show the flow of training data. Communication occurs via RPCs.

Durable , low-cost cloud storage

Composition

- **Dispatcher**

  a **centralized** dispatcher

- **Worker**

  a **dynamic** number of input data workers

- **Cache cluster**

  a **disaggregated storage** cluster for data caching

CacheW Overview

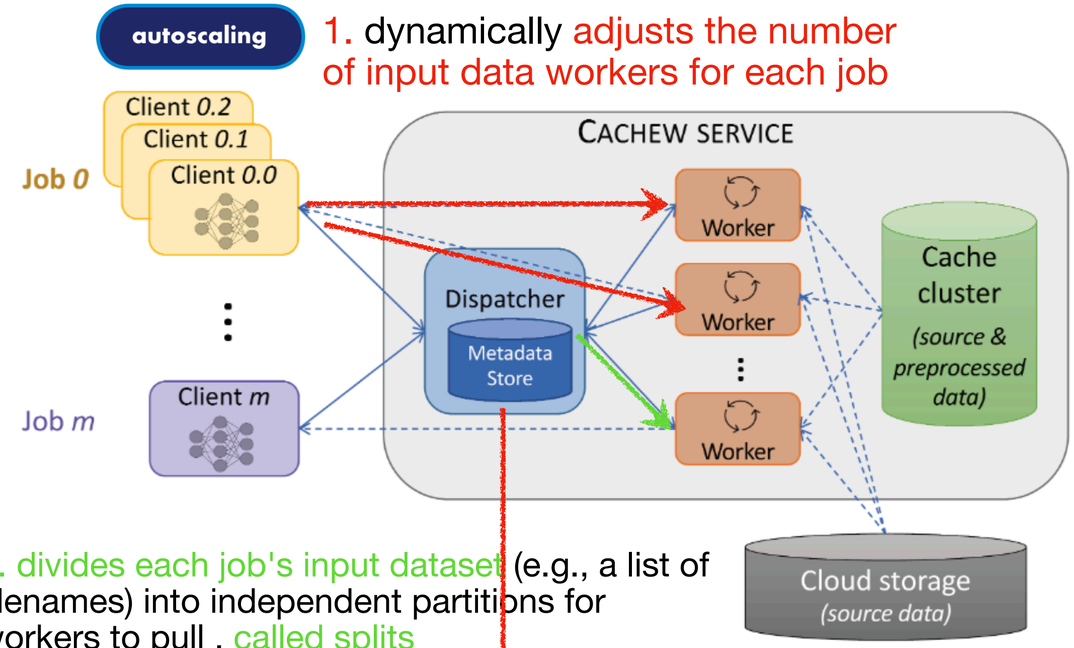


Figure 3: Cachew system architecture. Solid lines depict control logic and metadata communication. Dotted lines show the flow of training data. Communication occurs via RPCs.

splits may correspond to files that contain **already transformed (or partially transformed) data**

Composition

- **Dispatcher**

  stateless components responsible for **producing batches of preprocessed** data for clients (CPUs)

- **Worker**

- **Cache cluster**

CacheW Overview

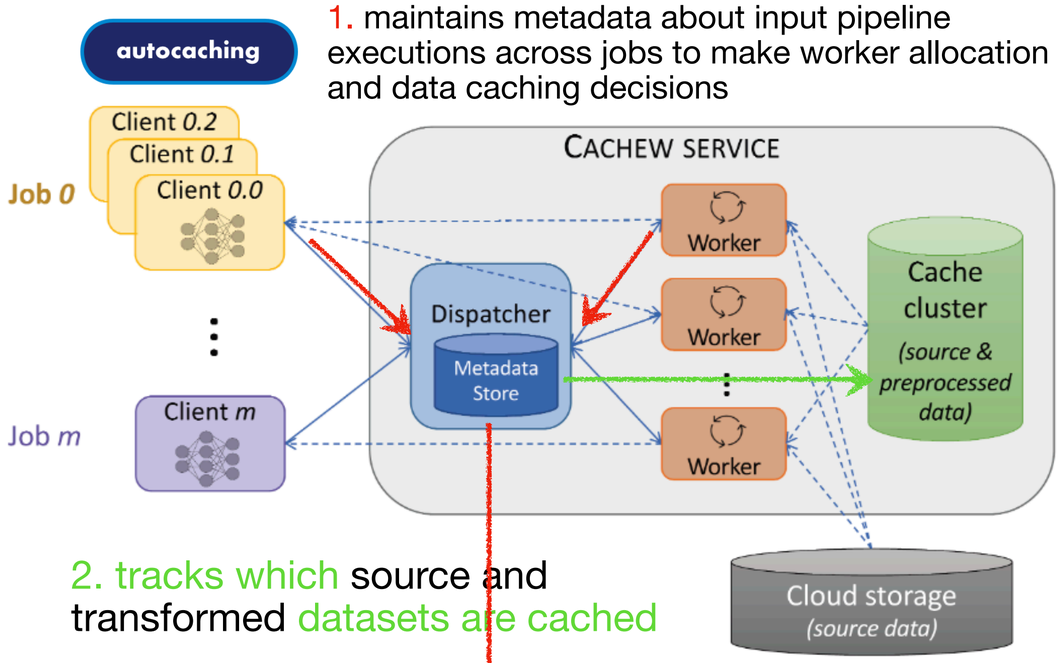


Figure 3: Cachew system architecture. Solid lines depict control logic and metadata communication. Dotted lines show the flow of training data. Communication occurs via RPCs.

Maintains the location of cached datasets in cache cluster — fingerprint

Composition

- **Dispatcher**

  stateless components responsible for **producing batches of preprocessed** data for clients (CPUs)

- **Worker**

- **Cache cluster**

CacheW Overview

1. Clients fetch data from the workers that are assigned to them by the dispatcher

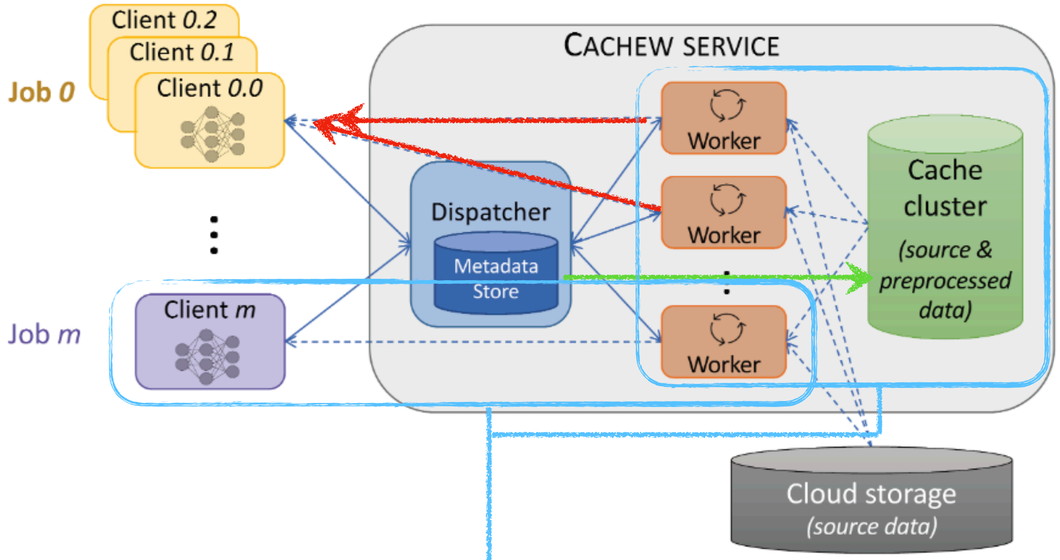


Figure 3: Cachew system architecture. Solid lines depict control logic and metadata communication. Dotted lines show the flow of training data. Communication occurs via RPCs.

Fully disaggregated -> fewer resource waste

Composition

- **Dispatcher**
- **Worker**
- **Cache cluster**
  disaggregate form worker — easy to **scale independently**

CacheW Overview

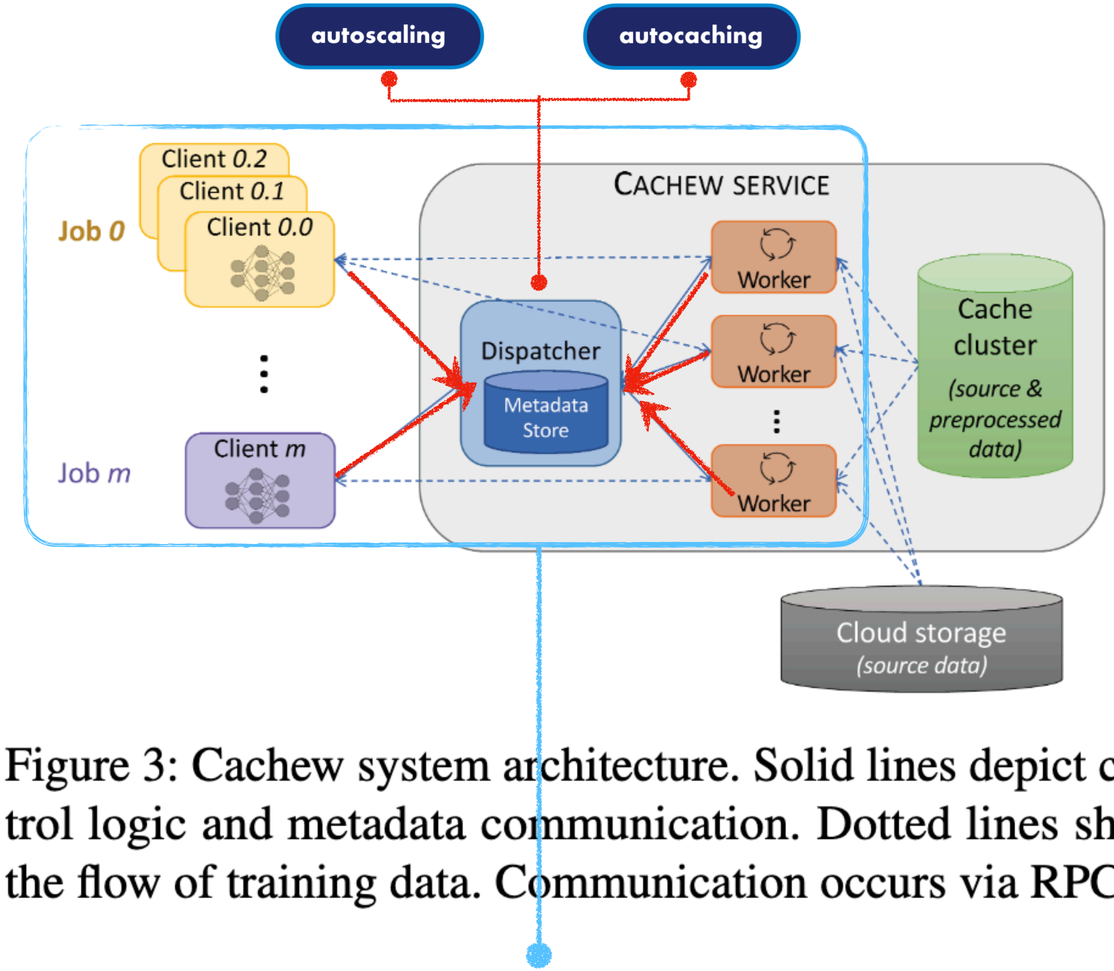


Figure 3: Cachew system architecture. Solid lines depict control logic and metadata communication. Dotted lines show the flow of training data. Communication occurs via RPCs.

Heartbeat to transform the metrics (every 5s)

Composition

- **Dispatcher**
- **Worker**
- **Cache cluster**

Source	Name	Description
Client	`batch_time`	Time taken to get and process the last 100 batches
	`result_queue_size`	Avg. number of batches located in the prefetch buffer over the last 100 batches
Worker	`active_time`	Avg. time per element spent in computation in the subtree rooted in the node
	`bytes_produced`	Total number of bytes produced by the node so far
	`num_elements`	Total number of elements produced by the node so far

Table 1: The set of metrics that are submitted by the workers and clients to the dispatcher via heartbeats.

CacheW API

A typical input data pipeline

1. dataflow operators can be parameterized with user-defined functions (**UDFs**)

2.autocache **before** randomly sampling will not always be applied

3. Communication between workers and clients are **abstracted** away from API

```
1  dataset = tf.data.TFRecordDataset(["file1", ...])
2  dataset = dataset.map(parse).filter(filter_func)
3            .autocache()
4            .map(rand_augment)
5            .shuffle().batch()
6  dataset = dataset.apply(distribute(dispatcherIP))
7  for element in dataset:
8    train_step(element)
```

autocaching

autoscaling

Figure 4: User API to distribute `tf.data` input pipeline execution with Cachew. Users insert `autocache` to hint which data is acceptable to cache/memoize and reuse within a job.

Autoscaling Policy — Dispatcher

How to leverages the per-job client metrics to make worker scaling decisions.

1. Allocate workers for new jobs

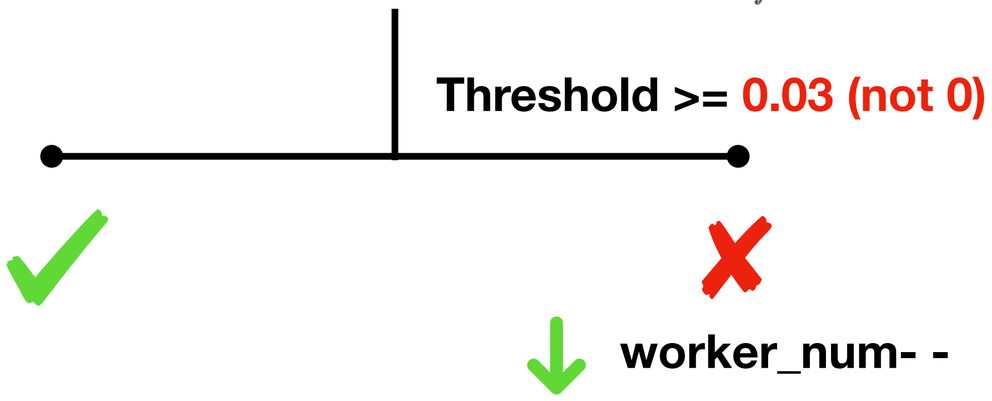
2. Re-scaling over time

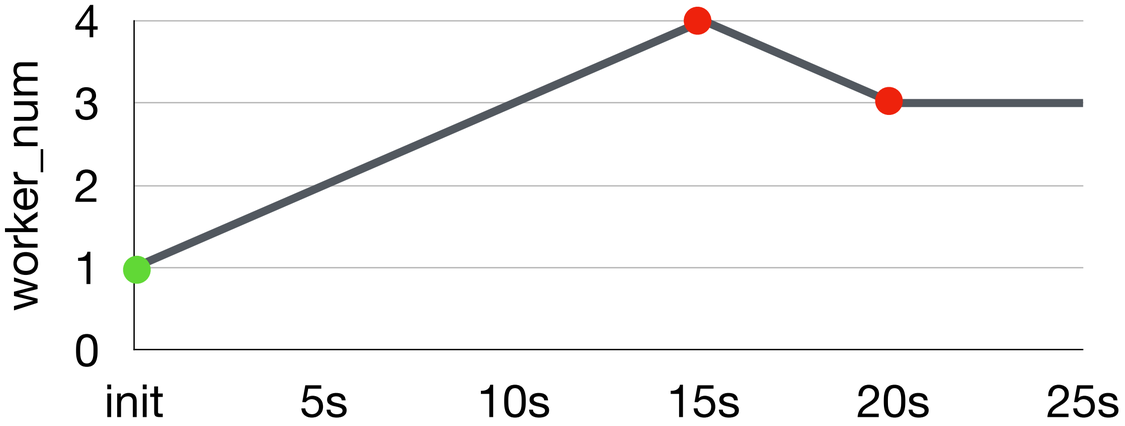
Autoscaling Policy

How to leverages the per-job client metrics to make worker scaling decisions.

1. Allocate workers for new jobs

$$speed\_up_{rate} = \frac{batch\_time_{aft} - batch\_time_{bef}}{batch\_time_{bef}}$$





Source	Name	Description
Client	`batch_time`	Time taken to get and process the last 100 batches
	`result_queue_size`	Avg. number of batches located in the prefetch buffer over the last 100 batches
Worker	`active_time`	Avg. time per element spent in computation in the subtree rooted in the node
	`bytes_produced`	Total number of bytes produced by the node so far
	`num_elements`	Total number of elements produced by the node so far

Table 1: The set of metrics that are submitted by the workers and clients to the dispatcher via heartbeats.

Threshold -> sensibility

Gather the metrics over more batches to reduce noise

Autoscaling Policy — Dispatcher

How to leverages the per-job client metrics to make worker scaling decisions.

2. Re-scaling over time

When : Every 10 new metrics received (1000 batches)

$$batch\_time_{current} >> batch\_time_{convergence}$$

↑ **worker_num++**

Source	Name	Description
Client	`batch_time`	Time taken to get and process the last 100 batches
	`result_queue_size`	Avg. number of batches located in the prefetch buffer over the last 100 batches
Worker	`active_time`	Avg. time per element spent in computation in the subtree rooted in the node
	`bytes_produced`	Total number of bytes produced by the node so far
	`num_elements`	Total number of elements produced by the node so far

Table 1: The set of metrics that are submitted by the workers and clients to the dispatcher via heartbeats.

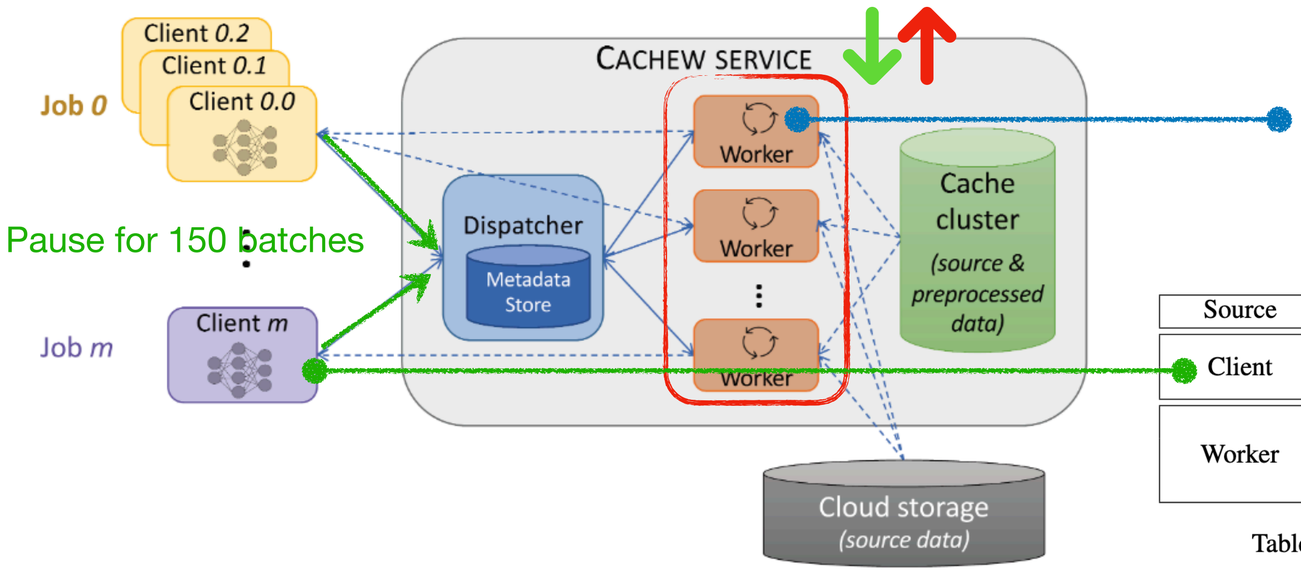
$$\frac{result\_queue\_size_{current} - result\_queue\_size_{convergence}}{result\_queue\_size_{convergence}} > 0.4$$

↓ **worker_num- -**

Autoscaling Policy — Dispatcher

How to leverages the per-job client metrics to make worker scaling decisions.

3. Ways to reduce noice



Source	Name	Description
Client	`batch_time`	Time taken to get and process the last 100 batches
	`result_queue_size`	Avg. number of batches located in the prefetch buffer over the last 100 batches
Worker	`active_time`	Avg. time per element spent in computation in the subtree rooted in the node
	`bytes_produced`	Total number of bytes produced by the node so far
	`num_elements`	Total number of elements produced by the node so far

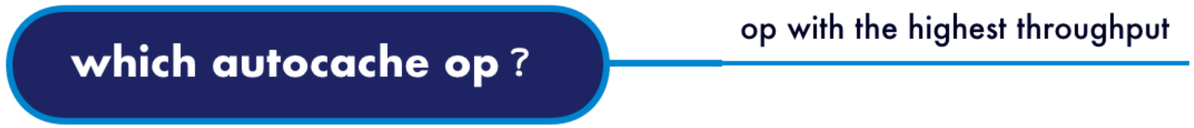
Table 1: The set of metrics that are submitted by the workers and clients to the dispatcher via heartbeats.

Figure 3: Cachew system architecture. Solid lines depict control logic and metadata communication. Dotted lines show the flow of training data. Communication occurs via RPCs.

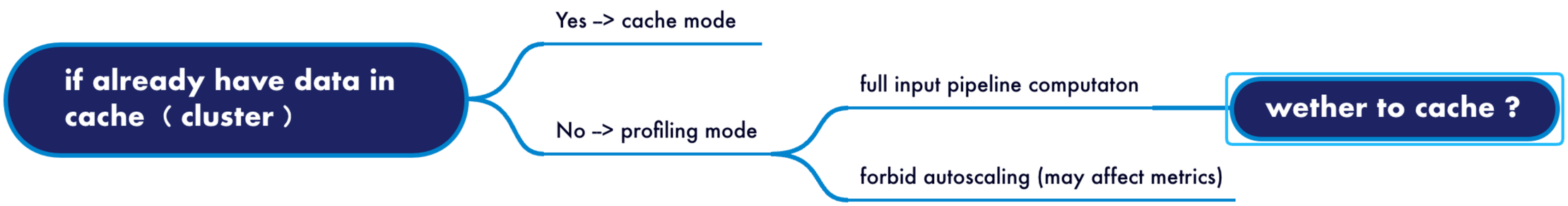
Model sees all data in an epoch, regardless of scaling events

Autocaching Policy — Dispatcher

Which to cache ?



Wether to cache ?



Autocaching Policy — Dispatcher

make comparison between time in cache & computing mode

$$TotalComputeTime = active\_time_{LastOp} \times N$$

$$PreAutocacheTime = active\_time_{autocache} \times M$$

$M \neq N$ Can be true

$$PostAutocacheTime = TotalComputeTime - PreAutocacheTime$$

$$ProjectedCachReadTime = M \times g_{GFS}(b_A)$$

$$PojectedTotalCacheTime = ProjectedCachReadTime + PostAutocacheTime$$

Throughout model — GlusterFS

Bytes per element at autocache op

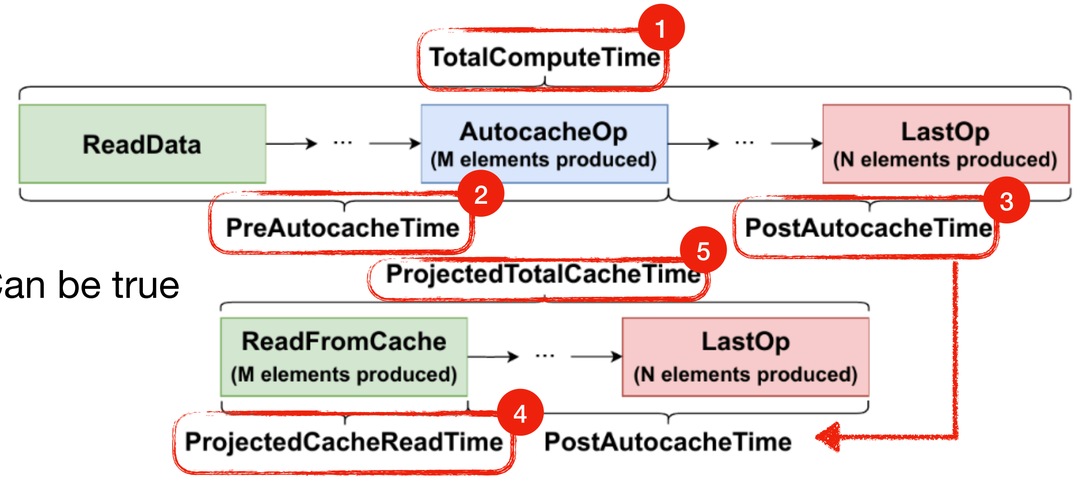


Figure 5: Cachew autocaching policy calculation.

Source	Name	Description
Client	`batch_time`	Time taken to get and process the last 100 batches
	`result_queue_size`	Avg. number of batches located in the prefetch buffer over the last 100 batches
Worker	`active_time`	Avg. time per element spent in computation in the subtree rooted in the node
	`bytes_produced`	Total number of bytes produced by the node so far
	`num_elements`	Total number of elements produced by the node so far

Table 1: The set of metrics that are submitted by the workers and clients to the dispatcher via heartbeats.

Fault tolerance

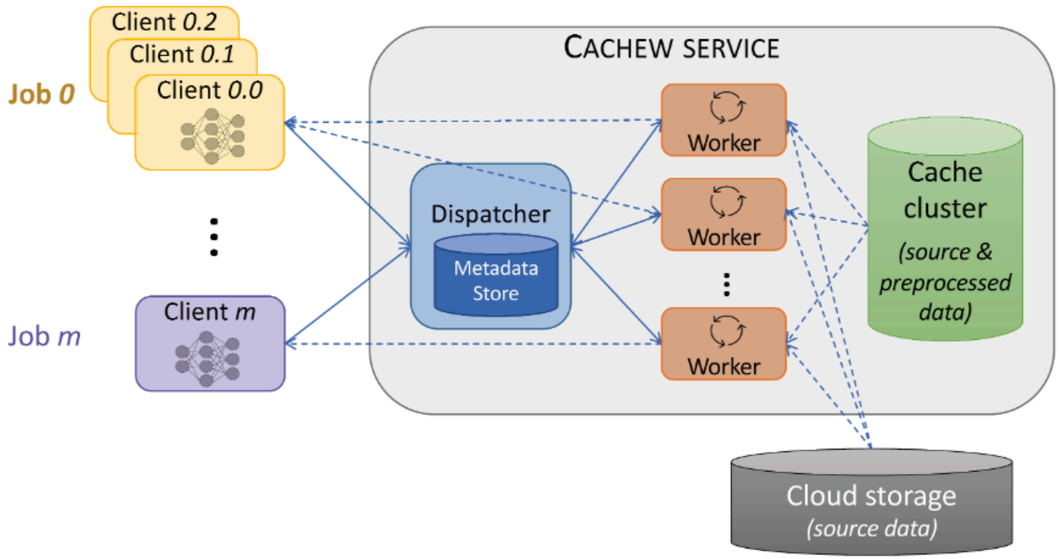


Figure 3: Cachew system architecture. Solid lines depict control logic and metadata communication. Dotted lines show the flow of training data. Communication occurs via RPCs.

Methods

- **Dispatcher**
  metadata for fast look-ups **(no state lost)**
  Journaling
- **Worker**
  replace by another (if miss heartbeats)
  restart from checkpoint (tf.train.checkpoint)
  transmit from breakpoint (**accuracy** & lower time)
- **Cache cluster**
  sufficient redundancy storage - 25%

Implementation

Autocache Mechanisms

1. Graph rewrites

- Rewrite input pipeline's dataflow graph
- Two version will be generated
  1. autocache -> put
  2. autocache -> get
- Dispatcher decides which version to use for workers

2. The put & get ops

- Build on snapshot op
- Used for store & retrieve data

3. Dealing with limited cache capacity

- Assume the cache capacity is unbounded (no limit for put op)
- Disuse useless data in cache cluster (Future work)

Evaluation

Experimental setup — workloads

WORKLOADS

model	dataset	Size
ResNet-50	ImageNet	2.6 x
RetinaNet	COCO	32.6 x
SimCLRv2	ImageNet	10.7 x

Data transformation increase data volume

Evaluation

Experimental setup —baselines

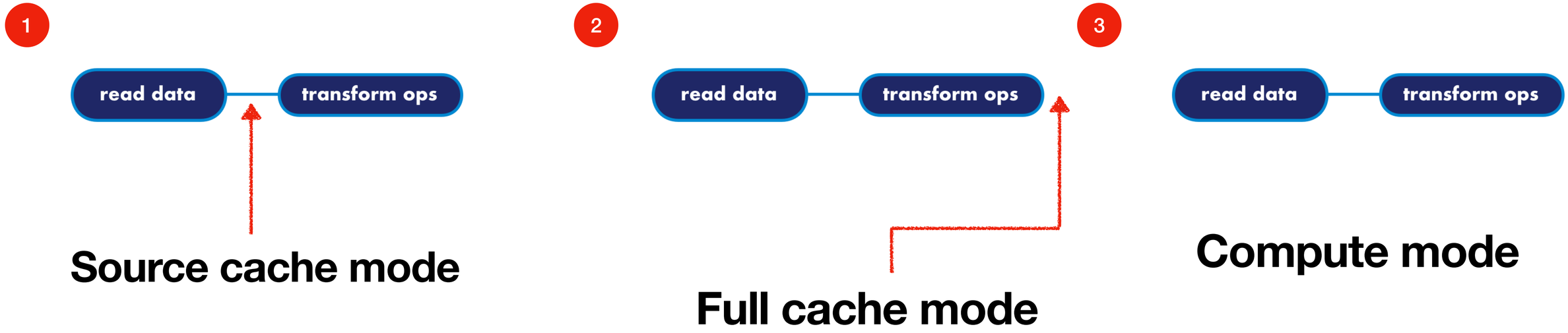
BASELINES

Kubernetes Horizontal Pod Autoscaler (HPA) -- scale based on **80% CPU resource utilization target** per node

infinitely fast input pipeline -- **no delay between workers and clients** (16 Gb/s)

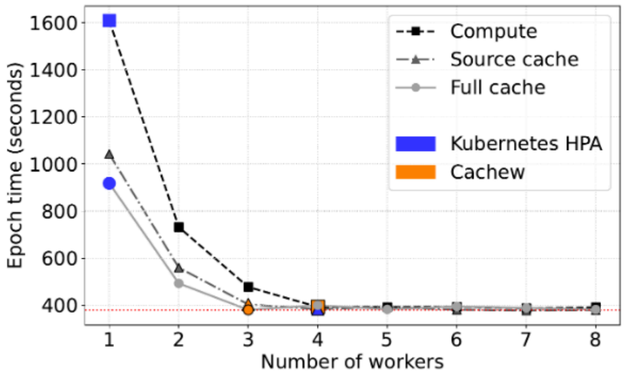
Evaluation

Experimental setup —execution modes (3)

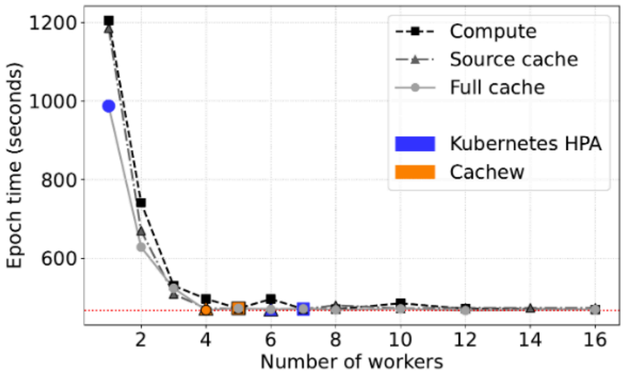


Evaluation

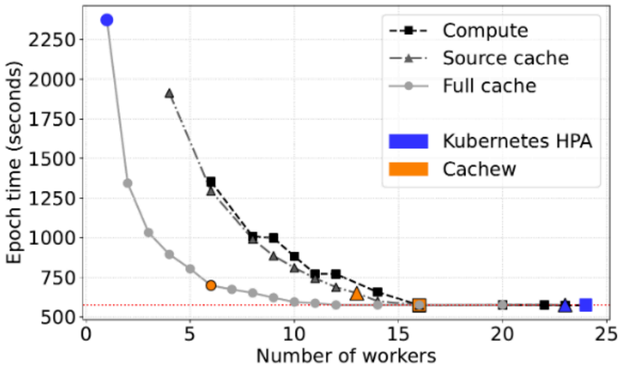
Autoscaling policy — compared with HPA



(a) ResNet50



(b) RetinaNet



(c) SimCLR

Figure 6: Scaling policy. Cachew selects the right number of workers to minimize epoch time and cost (orange markers). Kubernetes Horizontal Pod Autoscaler does not select the optimal number of workers (blue markers), since it only scales based on CPU usage and does not account for other potential input pipeline bottlenecks, e.g., memory and I/O bandwidth.

Cachew selects the **right number of workers** to minimize epoch time and cost

Evaluation

Autoscaling policy — sensitivity to threshold

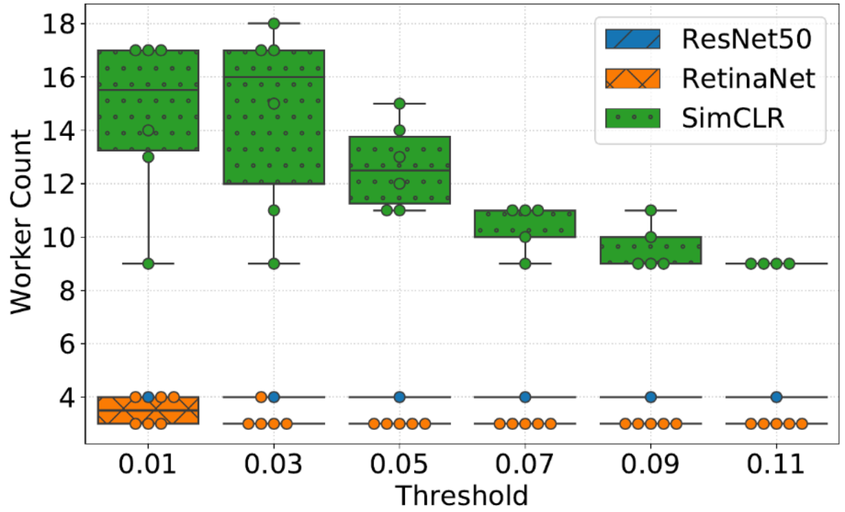


Figure 7: Cachew's first scaling decisions in compute mode relative to the value of the improvement threshold.

Gather metrics over multiple batches help to alleviate noise in **low threshold**

Evaluation

Autocaching policy

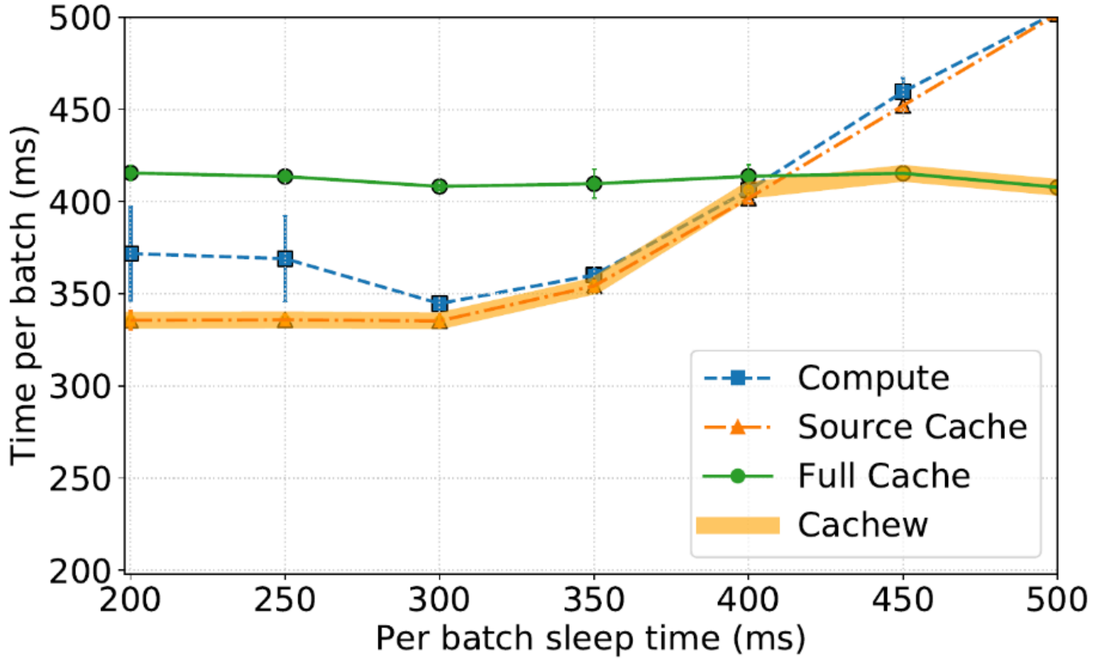


Figure 8: Cachew's autocaching policy selects the execution mode that minimizes batch time.

Evaluation

Autocaching & autoscaling over time

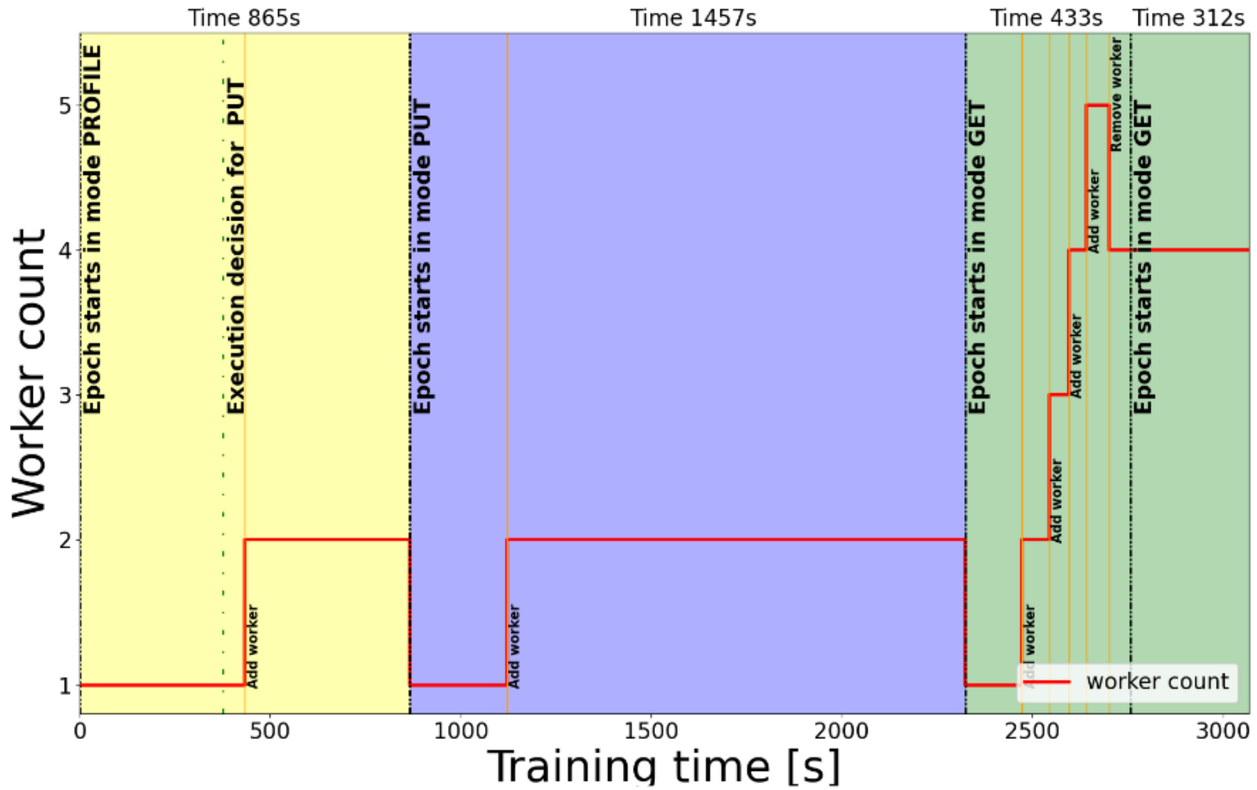
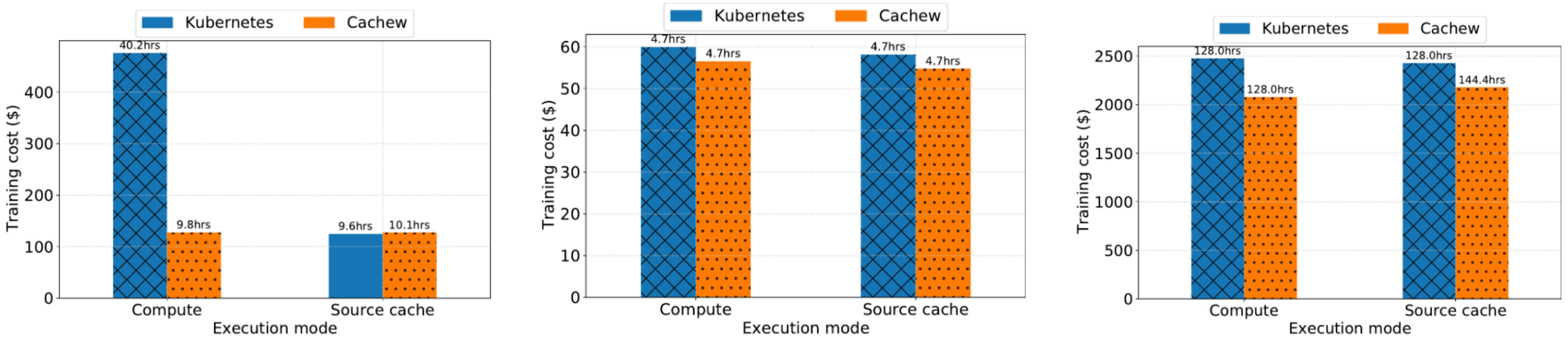


Figure 9: RetinaNet training timeline for the first 4 epochs. Cachew picks the right caching mode and number of workers.

Evaluation

End-to-end performance and cost



(a) ResNet50 (b) RetinaNet (c) SimCLR

Figure 10: Total training cost (and training time) for Cachew vs. Kubernetes HPA worker scaling policy decisions.

Reduce training time & cost **4.1x & 3.8x**

Conclusion

Advantage：

- **Jointly optimize input data processing by autoscaling & auto caching to avoid input data stalls**
- Fully disaggregated between work and client, worker and cache cluster bring flexibility

Disadvantage：

- Assume no delay between workers and clients, don’t consider network latency
- Can’t fix the information problem caused by cached transformed dataset (ATC’21 Alsys)
- Assume the cache capacity is unbounded

Thanks

2023-2-20

Presented by Guangtong Li